PEERCAN PROTocol

a simple peer-to-peer communication protocol for CAN bus.

## **HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | By | Description |
| 0.1 | 4/29/14 | Jonathan Guy | Initial draft |
| 0.2 | 5/8/14 | Jonathan Guy | Initial release into public domain |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## License

[[CC0](http://creativecommons.org/publicdomain/zero/1.0/)](http://creativecommons.org/publicdomain/zero/1.0/)  
To the extent possible under law, [Jonathan Guy](http://creativecommons.org/choose/zero/www.revely.com) has waived all copyright and related or neighboring rights to PeerCAN protocol. This work is published from: United States.

Although not required, the author requests reasonable attribution consistent with a Creative Commons license.

##### Trademarks

PeerCAN™ and the PeerCAN™ logo are trademarks of Revely Microsystems LLC

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet is a trademark of Open DeviceNet Vendor association, Inc.

## Introduction

PeerCAN is a simple, peer-to-peer network protocol for CAN bus. At a fundamental level, all PeerCAN data transactions occur using a system-wide set of data registers. The PeerCAN network is presented as a memory space containing 65535 addressable byte registers. Application software determines the format of the registers, allowing both simple and complex data exchanges depending on the system’s requirements. All PeerCAN operations are register-centric, rather than node-centric so therefore there is no need to uniquely address each node in the system.



The PeerCAN software stack can be implemented in almost any microcontroller or microprocessor that has a CAN2.0B controller, but the easiest solution to integrate is to use a dedicated PeerCAN controller. The PeerCAN controller connects via I2C or UART to the host MCU. This off-loads CAN bus management from the host MCU and makes PeerCAN a good fit for both simple and legacy embedded systems.

PeerCAN is architected to support UART-over-CAN and I2C-over-CAN with minimal additional configuration or software.

## BACKGROUND

CAN is a well-established multi-master serial network protocol, widely adopted in automotive and industrial applications. CAN is increasingly popular for use within systems – effectively filling the gap between I2C and Ethernet for multi-master communication. The challenge has been that most CAN protocols are moderately complicated – a necessity given the criticality of the target systems.

PeerCAN seeks to provide a simple application-defined way to control distributed I/O systems. It builds on a fundamental element of the CAN protocol: all CAN nodes receive and acknowledge every message on the bus.

The pre-requisites when defining the PeerCAN protocol were:

* Utilize the broadcast nature of CAN, where all nodes receive the packet and can generate an acknowledgement.
* Simple enough to replace discrete wiring within system (i.e. Super I2C).
* True Peer-to-peer communication so that a designated Master (or Masters) are not required.
* Data updates can be sent asynchronously, without the overhead of polling.

## FEATURES

* Uses CAN 2.0B and standard CAN transceivers
  + 29-bit identifier
  + High noise immunity
  + Up to 5000m bus length
* True Peer-to-peer.
  + No master needed. Data transfers occur directly between system nodes.
* Simple to learn
  + All operations use a simple hardware-centric register model.
* Easy to deploy
  + Adapts easily for differing system requirements
  + PeerCAN Modules accelerate development and integration

## COMPARISON TO OTHER PROTOCOLS

CANopen and DeviceNET are widely established CAN protocols. The following table generalizes the differences between the protocols.

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | PeerCAN | CANopen | DeviceNET |
| Architecture | Peer to Peer | Primarily Master/Slave  Also, Client/Server and Producer/Consumer | Master/Slave with Peer messaging to a limited extent |
| Typical Use | Intra-system communication | Field-installed systems | Field-installed systems |
| CAN Frame format | Extended frame only | Standard and extended frames | Standard and extended frames |
| Data encapsulation | Registers | Objects | Objects |
| Error Handling | At application level | Yes | Yes |
| Vendor ID license required | No | Yes |  |
| Specification pages | ~10 | 1000+ | 200+ |
| Maximum nodes | Only limited by bus characteristics | 127 | 64 |
|  |  |  |  |

## DATA FORMAT

Unlike other CAN protocols, PeerCAN does not have defined data types. PeerCAN data transfers are on the basis of the size of the data type, not the contents or range of values. For example, a single precision floating point number (4 bytes) and an unsigned 32 bit integer (also 4 bytes) are treated identically by PeerCAN.

Although the low-level protocol does not define or enforce data-types, the PeerCAN configuration file includes a data-type field for application level referencing. The possible types are INTEGER\_, UNSIGNED\_, REAL\_, and STRING. Integers may be 8, 16, or 32-bits. Real can be 32 (single-precision) or 64-bits (double-precision). There is no BOOLEAN data type as Boolean values are grouped and transferred as single or multi-byte integers at the application level.

PeerCAN is little-endian. That is, the least significant byte is stored at the lowest address.

Similarly, PeerCAN packets send the least significant byte as the first byte in the PeerCAN data field.

## TRANSFER

Normally data transfers are initiated by the Sender as determined by the application software. An event or timer might typically result in a register send. Receivers should always receive the register update and make it available to the application software on the receiver end. The default behavior is for the new register value to immediately update the old value. The PeerCAN protocol relies on the CAN frame ACK to acknowledge the transfer.

PeerCAN supports buffered transfers through the use of XON/OFF packets. Buffered transfers provide lossless transmission of data, and can be applied selectively to registers for transferring text, files, and other critical data.

## MESSAGE FRAME

The primary PeerCAN message is the register message. Register messages are initiated by a Sender and carry a payload of up to 8 bytes of data. Each packet also includes a 29-bit (extended) identifier. The identifier contains the destination register, data size information, and a PeerCAN CRC.

A PeerCAN message, aligned under a standard CAN packet, looks like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SOF | 11-bit identifier (ID) | SRR | IDE | 18-bit Identifier (ID) | | | | RTR | r1 | r0 | DLC | 0..8 Bytes of Data | CRC | ACK | EOF | IFS |
| SOF | Register Addres (11-bits) | SRR | IDE | Register Address (5-bits) | S1 | S0 | PeerCAN  CRC-11 | 0 | 0 | 0 | DLC | 1..8 Bytes of Data | CRC | ACK | EOF | IFS |

Figure 1 PeerCAN Register Message

The register address is located in the identifier to enable defined arbitration. The supported range is 1..65535. Every register has an intrinsic priority level based on its address, with lower addresses having the highest priority. If two PeerCAN transfers initiate simultaneously, the message to the lower-address will win arbitration while the CAN controller will automatically re-try the higher-address once the lower-address transfer is complete.

The recommended system configuration is to have only one sender assigned to each register address. In this configuration all messages will be sent to different addresses so there is no possibility of arbitration needing to extend past the first 16-bits of the ID field.

Multiple receivers can be mapped to a register, so broadcasts from a single node are supported without restriction.

PeerCAN does not prohibit two or more senders to send to the same register, so atypically, arbitration could continue into the S1/S0 field and beyond. At this point the CRC-11 field of the data values is highly likely to resolve arbitration between the two messages (the data payload is different). At this point there is a 1 in 1000 possibility that arbitration fails and a BIT ERROR is recognized. The CAN controller will generate an active ERROR frame at this point. Both nodes will then attempt to retransmit. This can lead to a non-ideal situation where the nodes repeatedly retransmit and then generate bit errors. The PeerCAN node should detect repeat errors as soon as possible (using a warning interrupt from the CAN controller error counter). If detected, each node should cancel the transmit operation and delay for a random interval (range TBD) before resending.

It is possible that two senders simultaneously send an identical message to the same register address. The message operation will complete with errors, which is both expected and preferred.

EID16 and EID17 specify the register width. This allows multiple registers updates to be included in a single message. This is typically only used for buffered registers. In non-buffered registers the subsequent value will over-write the previous value.

|  |  |
| --- | --- |
| S1, S0 (EID17, EID16) | Size |
| 0,0 | 1 byte |
| 0,1 | 2 bytes |
| 1,0 | 4 bytes |
| 1,1 | 8 bytes |

The CAN message data payload size should be an integer multiple of the EID-specified size.

The PeerCAN CRC-11 field is the CRC of the data payload and a pre-shared 16-bit key. The 16-bit key defaults to zero, which makes the authentication scheme optional. The CRC 11 uses the polynomial 0x583 (x^11 + x^9 + x^8 + x^2 + x + 1), which was selected from an excellent CRC polynomial table by Philip Koopman at CMU.

## OTHER MESSAGE FRAMES

### CONTROL MESSAGES

Control messages are sent to register address 0 using the standard extended frame format (RTR=0). All nodes should enable CAN controller ID filters to receive messages sent to register 0 (ID.13 to ID.28 are all zeroes).

Using Register address 0 for control messages prioritizes control messages over all other PeerCAN messages.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SOF | 11-bit identifier (ID) | SRR | IDE | 18-bit Identifier (ID) | | | | RTR | r1 | r0 | DLC | 0..8 Bytes of Data | CRC | ACK | EOF | IFS |
| SOF | 0 | SRR | IDE | 0 | S1 | S0 | PeerCAN  CRC-11 | 0 | 0 | 0 | DLC | 2 Data bytes representing the register address | CRC | ACK | EOF | IFS |

Figure 2 PeerCAN Control Message

Depending on the number of filters available in the CAN controller, some nodes will need to handle all control messages in software. However the software overhead in servicing control messages is typically very low and can be completed with a few bit test operations.

The S0, S1, and data payload field definitions change from the standard PeerCAN format.

|  |  |  |
| --- | --- | --- |
| S1, S0 (EID17, EID16) | Function | Data Payload |
| 0,0 | XOFF – stop sending data to FIFO register | Register FIFO for XOFF action (1..65535) |
| 0,1 | XON – start sending data to FIFO register | Register FIFO for XON action (1..65535) |
| 1,0 | Register request | Specifies the Address of the register being requested (1..65535). |
| 1,1 | Extended control message | See extended control message definition |

##### FLOW CONTROL FOR FIFO REGISTERS

XON and XOFF control is initiated by the receiver when the receive buffer is full or almost full. The threshold for an XOFF message is normally when the buffer has space for 8 or fewer bytes. This allows for one additional PeerCAN register message to be received before the XOFF message is sent. XOFF must be queued in the CAN controller before the end of the additional PeerCAN register.

The data payload is the 16-bit register address of the FIFO register. The sender will use this value to identifier the buffer stream for action.

Upon receiving an XOFF frame, the sender should cease transmission to that register until an XON message is received. The default state is XOFF, so an XON message is required to initiate communication.

##### REGISTER REQUEST

The Register request message is used by a node to request a register update from the sender of a register. The data payload is the 16-bit register address being requested. While register requests are supported, asynchronous transfers should be used whenever possible to reduce bus traffic and unnecessary overhead on nodes.

##### EXTENDED CONTROL MESSAGE

Extended control messages are normally used only for debugging. The extended control messages support the transfer of register configuration information, so that a debug device can observe and understand the entire network.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data byte | 0 | 1 | 2 | 3 | 4 |
| Function | Register Address High-byte | Register Address Low-byte | Control Code | Receive Address (High-byte) | Receive Address (Low-byte) |

Control codes are:

|  |  |
| --- | --- |
| Code | Action |
| 0 | Terminate request/action |
| 1 | Request register configuration (omit optional fields) |
| 2 | Request register configuration (all fields) |
| 3 | Request node configuration (omit optional fields) |
| 4 | Request node configuration (all fields) |
| 5..255 | Reserved |

Control codes 1..4 request a transfer of the configuration information, formatted as an ASCII text message to the specified send-to FIFO. These are temporary requests and the FIFO channel is closed once the transfer is complete. Multiple simultaneous configuration requests may be initiated but the node/register is not required to fulfill the request.

PeerCAN JSON format is used, although support of this extended message is optional. Nodes or registers that do not support this message should respond by sending EOF (ASCII 0x04) to the specified FIFO receive register. An EOF can also be sent if a node or register is busy handling another request.

## VARIANCES FROM CAN 2.0 SPEC

PeerCAN has the following variances from the CAN specification:

* The requirement that the first 7 bits of the identifier not be dominant (‘1’) is ignored to allow full 16-bit address range. This requirement was a limitation of early CAN controllers made by Intel and is not relevant to PeerCAN. Never-the-less, PeerCAN compatibility with a CAN controller should be confirmed.

## REGISTERS

The entire 16-bit address range is available for register allocation except for address 0 which is reserved for XON/XOFF commands.

Although it’s permissible to randomly assign registers inside the 1..65535 address range, the preferred method is to group the registers based on the receiver. That is, any registers that a node is receiving should be in a continuous block to allow the CAN controller to effectively apply message filtering. This substantially reduces the overhead of interrupting on every CAN message.

Up to 3 blocks of registers are defined, each of which has a separate ID mask applied by the CAN controller. Having 3 blocks prioritization of each block (high, medium and low) allows relative to each other and, more importantly, to other messages in the system.

Optimally, allocated register blocks should start at a 2n boundary (i.e. 256) and finish at 2(n+1)-1 (i.e. 511) so that the CAN controller can fully mask any non-relevant CAN operations. Practically, the PeerCAN stack should accommodate blocks starting at any location, even if the ID mask requires a second level of software-based filtering.

## PEERCAN CONFIGURATION

A PeerCAN definition file has two sections. The first defines a few key parameters for the node. The second section defines the register configuration.

Node configuration fields:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Field | NodeName | SystemKey | BitRate | SamplePoint |  |  |
| Description | Name for this node (optional) | 16-bit value unique to the system | Bit rate code (see table) | Sets the CAN bit sample point in 16th of a bit time. |  |  |
| Default | NULL | 0 | 8 | 14 |  |  |
| Range | STRING[40] | 0..65535 | 1..8 | 8..15 |  |  |

Structure size in bytes = 44

|  |  |
| --- | --- |
| Bit rate | Code |
| 1 Mbit/s | 8 |
| 800 kbit/s | 7 |
| 500 kbit/s | 6 |
| 250 kbit/s | 5 |
| 125 kbit/s | 4 |
| 50 kbit/s | 3 |
| 20 kbit/s | 2 |
| 10 kbit/s | 1 |

Register configuration fields:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Field** | NodeReg | NodeRegName | Comment | Size | Type | Format |
| **Description** | Local register number | Local register name (optional) | Comments or notes (optional) | Node register size in bytes | Sets data direction and buffering | Specifies data type (optional) |
| **Range** | 0..65535 | STRING[24] | STRING[40] | 1..8 | SEND  RECEIVE  RECEIVEACTIVE  SENDBUFF  RECEIVEBUFF | INTEGER\_, UNSIGNED\_, REAL\_, and STRING. Integers may be 8, 16, or 32-bits. Real can be 32 (single-precision) or 64-bits (double-precision). |

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | SystemAddress | SystemName | SystemComment |
| **Description** | System Register Number | System Register Name (optional) | System Register Comment (optional) |
| **Range** | 0..65535 | STRING[24] | STRING[40] |

Structure size in bytes (minimal, per register) = 6

Structure size in bytes (full, per register) = 135

The definition file uses JSON, a light-weight data-interchange format. The JSON file can be used directly on a node (if a basic file system is present), converted into another data format, or converted to an include-type file for compile-time integration with a node. The JSON file must be limited to ASCII characters, as UTF-8 encodings are not supported.

{

"NodeName":"A name for this node",

"SystemKey":0,

"BitRate":8,

"SamplePoint":14,

"Register":

[

{

"NodeReg": 100,

"NodeRegName":"Reg name: Node context",

"Comment":"Any additional comments or description",

"Size":2,

"Type":"SEND",

"SystemAddress":512,

"SystemName":"A name in the context of the system",

"SystemComment":"Any additional comments or description"

}

]

}

## Exceptions

At this time error handling is not a defined part of the PeerCAN protocol. The intention is that all exceptions are handled at the Application layer.

# OTHER GUIDELINES

The following guidelines are not part of the formal PeerCAN specification but are provided as guidance for implementation PeerCAN systems.

## PEERCAN TRANSCEIVERS

PeerCAN recommends the use of +3.3V CAN transceivers, although +5V transceivers are acceptable. +3.3V transceivers offer comparable bus and noise performance, but achieve an almost 50% reduction in transceiver power consumption (see TI SLLA337 application report). Both 3.3V and 5.0V transceivers are interoperable as the fundamental voltage thresholds on the CANH and CANL signals remain the same.

Since 5.0V power rails are increasingly uncommon in embedded system, 3.3V operation is better suited to the applications targeted by PeerCAN.

## PEERCAN WIRING

PeerCAN bit rate and bus length specifications are identical to established parameters from the CANopen protocol. The default sample point should be as close as possible to 87.5% of the bit time (7/8th of the bit time).

|  |  |  |  |
| --- | --- | --- | --- |
| Bit rate | Bus length | Max. stub length | Accumulated stub length |
| 1 Mbit/s | 25 m | 1,5 m | 7,5 m |
| 800 kbit/s | 50 m | 2,5 m | 12,5 m |
| 500 kbit/s | 100 m | 5,5 m | 27,5 m |
| 250 kbit/s | 250 m | 11 m | 55 m |
| 125 kbit/s | 500 m | 22 m | 110 m |
| 50 kbit/s | 1000 m | 55 m | 275 m |
| 20 kbit/s | 2500 m | 137,5 m | 687,5 m |
| 10 kbit/s | 5000 m | 275 m | 1375 m |

## I2C OVER PEERCAN

Although not part of the formal protocol, PeerCAN is designed to support I2C operations over PeerCAN.



The exact implementation is left up to the PeerCAN node, but essentially I2C read and write operations are mapped into the PeerCAN register space. To allow for the latency of the PeerCAN operation, the PeerCAN slave should implement clock stretching until the PeerCAN operation is complete. For I2C writes, the completion event is a successful CAN ACK. For I2C reads, the read can either be handled using the most recent value sent from the remote node, or by initiating a register read operation.

# CONCLUSION

The PeerCAN protocol has defined a simple method for using extended CAN frames for simple peer-to-peer communication in an embedded system. The intention is to offer a solution that fills the gap between I2C and more complex communication networks.