APPLICATION NOTE

AN245
Using the PDI1394L11 “AV Link” Controller

Author: Allen Light and Rudi Bloks

Sep 10, 1997

Philips
Semiconductors

PHILIPS

Application Note

Using the PDI1394L11 "AVLink" Controller

Allen Light and Rudi Bloks

Sept. 10, 1997

Philips Semiconductors
Application Specific and Standard Products

1 Introduction

IEEE 1394-1995 is a standard for a high speed serial interface bus. The protocol stack consists of a physical layer (PHY), a link layer (LINK) and a transaction layer (TRANS) and of course there is always a management component to control all layers. The standard defines several packet types at the link layer level and several transaction types and also defines how some of these must be used for management purposes. It does not define how applications can or should use these transactions and packet types to exchange data. See ref. [1] and [2].

IEC-61883 (ref. [3]) is a draft for a standard that will ensure interoperability amongst consumer electronics devices which use 1394 as an interface bus. It defines the method to be used to pack several audio and video related data types into 1394 packet payloads (e.g. MPEG-2, DVC-SD/HD) and it also specifies procedures and resources to establish and manage virtual connections between these devices for these data streams as well as procedures and resources for command and control function (Function Control Protocol and Command Transaction Sets).

The PDI1394L11 enables easy implementation of the IEEE 1394 interface bus in consumer electronic devices that need to be IEC-61883 compliant. The PHY layer is implemented as a separate IC, (PDI1394P11) allowing it to be exchanged for a higher speed version later (when available). The LINK layer, part of the TRANS layer and parts of the IEC-61883 "AV" layer are implemented in the AVLink chip. The remaining parts must be handled in software (e.g. on the CPU of the host system).

It is assumed that the reader of this document has a certain basic knowledge of 1394 higher levels and access to both standard specifications mentioned here (IEEE 1394 and IEC-61883).

This document is meant to provide some useful background information on the AVLink chip, in particular the use of all the functionality offered and how to program various parameters, as seen from a users point of view. For an overview of 1394 tasks to perform (1394 bus management) refer to [5].

Note: This document primarily describes the functionality of the latest design for AVLink, which includes several changes since the creation of the first silicon. Chapter 10 describes the differences between the first silicon and the latest design.

2 Isochronous data transmissions for AV devices

2.1 Challenges with 1394

There are many different audio/video data types in existence today (MPEG, DVC, DSS, etc.) and they all have in common that they use packetized transfers, e.g. data is grouped in units of constant size that usually start with some header. The number of packets per second and the size of these packets can vary over a very wide range depending on the data type. On 1394 there is only one type of isochronous data packet and only one packet may be transmitted per channel per isochronous cycle (= 8000 packets per second per channel). Since channels are used to represent connections between devices this leaves a maximum of 8000 packets per second for any specific connection. How to map AV data to 1394 isochronous packets?

Before any isochronous transmission can be done bandwidth must be allocated for it. This bandwidth is measured in the number of bit cells at some specific speed that the transmitter wishes to use in each isochronous cycle (i.e. 8000 times per second). If this amount varies from cycle to cycle then the peak value must be allocated and this amount will then actually be available in every cycle. Clearly it is important to smooth out the actual use of the bandwidth as much of possible to get the peak value down (allowing more channels to operate simultaneously).

Another problem with 1394 is that the arbitration protocols introduce a considerable jitter (variation) in the transport delay of subsequent packets belonging to a single stream. Some data types such as MPEG are very intolerant and require a constant transport delay, or at least a jitter far less than what 1394 can offer. Some solution is required to enable data transport with a constant delay across 1394.

2.2 Solutions offered by IEC-61883

IEC-61883 addresses these problems in the following way (see also [4]):

- To enable reconstruction of original application packet timing at the receiving side a transmitter can attach a time stamp to each data packet received from the application. This stamp is a 25 bit sample taking from the Cycle Timer Register of the local LINK (least significant 25 bits) incremented with some offset (delay) value, and represents the intended delivery time of the packet to the host application at the receiving node. The stamp is stored in the lower 25 bits of a quadlet whose upper 7 bits are zero and this quadlet is inserted in front of the corresponding application packet.
- To accommodate various bandwidth requirements and packet sizes, the application data packets (with the optional 4 byte time stamp added) are divided into a number of equally sized fractions called data blocks. The number of fractions can be chosen from 4 values: 1, 2, 4 or 8 and is a defined constant for a given data type (i.e. it should not vary from packet to packet).
  Since not all packets can be perfectly divided into 2, 4 or 8 data blocks some dummy quadlets (zero-filled) may have to be added at the end of the original application packet to get a size which is a multiple of 2, 4 or 8. This is called quadlet padding. The number of quadlets to pad never exceeds or equals the number of data blocks into which packets are divided and it also never exceeds or equals the size of one such data block. As a result only the last data block of any application packet can contain padding quadlets.
  Data blocks are then used as elementary (smallest) units of data to construct bus packets. The number of data blocks to be sent in a bus packet depends on the allocated bandwidth and data type specific alignment and transmission criteria, to be defined in IEC-61883. This mechanism allows a single application packet to be transferred over many isochronous cycles (for low bit rates) but also to transfer many data

blocks, and thus multiple application packets, in a single isochronous cycle (for high bit rates) and anything between these extremes.

- Various parameters used to pack the original data packets at the transmitter are conveyed in the stream itself to any receiving node to allow that node to unpack the stream and regenerate the original application packet (and their timing if applicable) without special prior knowledge of the data type, i.e. it is possible to build a generic receiver that can handle future data types. The information is stored in a special 2 quadlet header that is located at the start of every isochronous bus packet payload (called a CIP header: Common Isochronous Protocol). This header also contains a data type indication (called ’format’), format dependent flags, a sequence number and an optional high level application synchronization time stamp (do not confuse with transport delay time stamp).

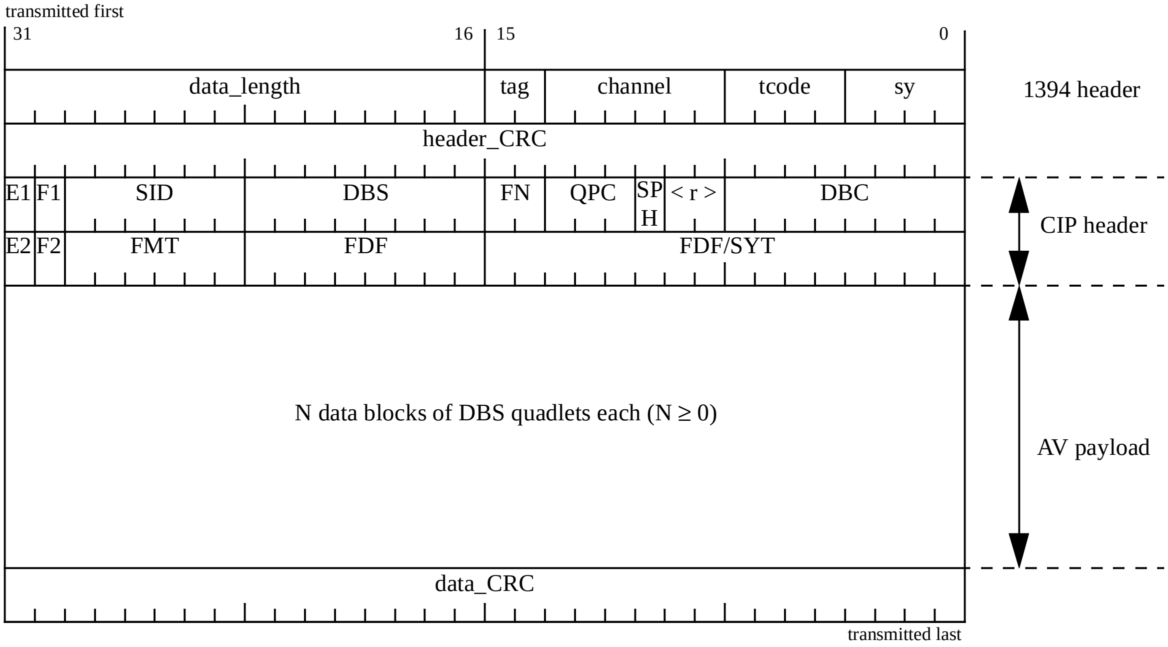


Figure 1: Format of IEC-61883 isochronous packets on the 1394 cable.

The fields of the 1394 header are defined in the IEEE 1394-1995 specification:

The *data_length* field of the 1394 header contains the number of bytes in the payload field, which equals 4 * (2 + N * **data_block_size**) and is thus always a multiple of 4. The **data_block_size** in this formula is the size of each data block measured in quadlets ranging from 1 to 256.

The *tag* field has a constant value allocated by the IEEE, ’01b’.

The *channel* field contains the number of the logical channel for this connection.

The *tcode* field is fixed (defined by IEEE-1394) to value ’1010b’.

The *sy* field is not defined (for application use).

The fields of the CIP header are defined in IEC-61883:

The *F* flags (*F1* and *F2*) are 0 if this format is used, or 1 if another format (TBD) is used.

The *E* flags indicate the end of the header (so *E1* = 0 and *E2* = 1).

The *sid* field contains the 6 bit physical ID of the transmitting node on the local 1394 bus.

The *dbs* field contains the **data_block_size** (value 0 means 256 quadlets).

The *fn* field contains a value that indicates the number of fractions into which the original application packet was divided (after optional stamp and padding were added): 00 = 1, 01 = 2, 10 = 4, 11 = 8. This can be expressed as a mathematical relation as well: number of data blocks = 2^{fn}.

The *qpc* field contains the number of quadlets added as padding at the end of each application packet to make it divide nicely into 2^{fn} data blocks (note that 3 bits are always sufficient).

The *sph* field indicates whether time stamps are present (added by transmitter).

The bits marked <r> are reserved and should be zero (0).

The *dbc* field acts as a sequence counter for continuity checking and increments by one for every data block transmitted on the cable. The value in the CIP header is the sequence number of the first data block following the CIP header.

The *fmt* field indicates the data format (e.g. MPEG, DVC,...) in the stream.

The *fdf* field contains format dependent flags (defined separately for each data type) and can be used to carry additional (static or very slowly changing) information about the data in the stream. Optionally the lower 16 bits can be used to carry a synchronization time stamp (*syt*) for application level synchronization. It’s use is application specific and defined in IEC-61883 if applicable. This stamp, if present, is created in a manner similar to the packet delivery time stamps (based on Cycle Timer Register).

Note that no information is carried about byte padding. Application packets are supposed to be a multiple of 4 bytes in length. If they are not for some new data type then an extension of this IEC-61883 protocol may be required.

It is convenient to assign a name to the intermediate packets that exist in the IEC-61883 layer after adding an (optional) time stamp and quadlets of padding (if required). These packets form the source from which bus packet payloads will be constructed by dividing them into fractions that are used as building blocks for these payloads. These intermediate packets shall be referred to as *source packets*.

To allow receivers to easily resynchronize to a stream after missing an entire bus packet or after tuning in to an already existing stream there is an additional requirement that holds for all data types: the first data block of any source packet has a sequence number S for which the following condition must hold:

$$S \ \mathbf{mod}\ 2^{fn} = 0$$

Using the *dbc* and *fn* field values from the CIP header the receiver can easily locate (count down to) the start of the next source packet in the incoming data stream.

Every active isochronous IEC-61883 transmitter must send a packet during every cycle even if there is no data to transmit. In that case an empty CIP packet is transmitted which contains only a CIP header in the payload field and no data blocks. The values in the CIP header are the same as what they would have been if the payload had not been empty. In some cases a transmitter may not be able to gain access for isochronous transmission, for example when a bus reset is taking place, or when the cycle start packet was missed due to a bit error on the cable. In such cases the transmitter may discard the entire payload it would have sent otherwise (resulting in a sequence error at the receiver when transmission resumes in the next cycle).

2.2.1 MPEG-2 Transport Stream (IEC 61883 specification)

An MPEG-2 TS uses transport packets (TP) with a length of 188 bytes, or 47 quadlets. Since MPEG is time critical delivery must be time stamp based: *sph* = 1. The resulting size is 48 quadlets which divides nicely so no quadlet padding is required: *qpc* = 0. To allow efficient use of bandwidth even at low bit rates (1.5 MBit/s) the number of data blocks is fixed at 8 (*fn* = 3) and the data block size is therefore 6 quadlets (*dbs* = 6). The value of *fmt* for MPEG-2 is defined to be 32 and the *fdf* field is completely reserved (*syt* is not used). See also [4].

The 8 data blocks in a source packet cannot simply be stowed into bus packets without restrictions:

source packet:	db0	db1	db2	db3	db4	db5	db6	db7

The construction of bus packet payloads is restricted in the following manner:

- For very low bit rates requiring at most 1 data block per cycle (R < 1.5 Mbit/s) any of db0..db7 can be in the payload (i.e. no restriction).
- For low bit rates requiring at most 2 data blocks per cycle (1.5 < R < 3 Mbit/s) a payload contains either 2 consecutive data blocks starting with an even number (db0/1 or db2/3 or db4/5 or db6/7) or it is empty (CIP header only). If only one data block is available for transmission it will not be sent!
- For medium bit rates requiring at most 4 data blocks per cycle (3 < R < 6 Mbit/s) a payload contains either 4 consecutive data blocks (either db0/1/2/3 or db4/5/6/7) or it is empty.
- For higher bit rates a payload always contains one or more entire source packets.

Since bit rates in MPEG transmissions can change mid-stream it must be possible to switch between the above ’modes’ while a transmitter is active. It is the transmitter’s responsibility to ensure that the above alignments are always correctly applied.

2.2.2 DVC-SD/HD (IEC 61883 specification)

DVC uses a raw data packet called a DIF block which is 80 bytes long. For SD (standard definition) 6 of such DIF blocks together form one application packet. For HD (high definition) 12 blocks are grouped. The mapping is simple in this case: there is no time stamp required (*sph* = 0) and no padding (*qpc* = 0) since there is only one data block per source packet (*fn* = 0) and therefore the size of the data block is the size of the application packet (*dbs* = 120 quadlets for SD, 240 for HD).

The *fmt* field value is 0 and *fdf* contains a few flags for signal identification (such as frame rate). The *syt* field is used to send a low frequency frame synchronization pulse that can be used for servo sync during track-to-track copy.

A bus packet contains either one data block (= one application packet) or just a CIP header. The bandwidth requirements are fixed and thus cannot change midstream.

3 Asynchronous data exchange

For data without real-time requirements, such as command and control functions, IEEE-1394 offers a type of data exchange called *asynchronous transactions*. A transaction typically involves the transmission of a *request* packet from the originating node (the requester) to the target node (the responder), which is optionally (depending on the type of request) followed by a *response* packet going back from the responder to the requester. The types of requests and responses are all defined in IEEE-1394.

At the application level in a 1394 device (i.e. on top of the transaction layer) a transaction is done simply by presenting a request packet to the transaction layer for transmission and waiting for the response packet (if any). One level below, at the link level every primary asynchronous packet (including all transaction packets) requires an short acknowledge code in return from the destination node, unless the packet was broadcast to all nodes on the bus (in which case there is no single destination node). If the acknowledge code contains a *busy* indication then the destination node for the packet could not handle the packet at this time and it may be retransmitted again later for a retry. Such retransmissions are handled inside the transaction layer.

The response packet is not necessarily sent immediately following reception of a request. In fact most nodes won’t be able to create a response fast enough in their transaction layer or application layer. For such cases the link layer may be instructed to return a special *pending* acknowledge immediately after reception of the request packet. This code informs the originator that a proper response shall follow at a later time (within a specified time out, default 100 msec.). The transaction layer is supposed to track these pending responses and generate time-out indications to the application layer if a response does not follow within the specified time interval.

There are a few other acknowledge codes that may be returned by a destination node’s link layer, all of which are error indications that will cause a premature termination of the entire transaction. See IEEE-1394 for a full definition.

The type of operations provided by the three categories of transactions (read, write and lock) are sufficient to allow just about any kind of non real-time data exchange, including broadcast messages. Consequently there was no need to further enhance 1394 with special operations at this level for use in consumer electronic devices. However, 1394 does not define *how* these transactions should be used for functions that are required to obtain interoperability between different devices of different manufacturers with similar functionality, e.g. VCRs or set-top boxes.

The same IEC-61883 draft that defines the isochronous data formats also defines several resources (Control and Status Registers (CSRs) placed in the 1394 bus architecture) as well as rules for accessing and modifying these CSRs using standard transactions, and the effects of such modifications on the devices or data streams they control. These protocols are setup such that they imply a growth path: The simplest device (cheap) needs to implement only a bare minimum of functionality, just enough to let it start sending isochronous data or listen to isochronous data on a predefined (default) channel. On the other end of the scale there could be a complex system controller capable of setting up and maintaining all kinds of data connections between several devices simultaneously and synchronizing applications through asynchronous commands and using an advanced user interface.

There are several CSRs which are mandatory for any 1394 node, some which are optional for 1394 but mandatory for IEC61883 and some that are new (either optional or mandatory) in IEC61883. CSRs can be implemented in hardware as well as software. Each CSR definition clearly states what types of transactions are allowed and what their effects are. A hardware implementation also requires all logic needed to carry out those request operations and compile an answer in a response packet. A software implementation requires code to examine incoming packets and see if they access one of the CSR locations in the 1394 bus model architecture, then perform the required operation and generate a response packet. This is probably cheaper to implement but also slower, which could become a problem for manager nodes in large networks. Combinations are also possible (some CSRs, or some types of transactions in hardware, others in software).

Some of the operating parameters for the transaction layer and link layer are defined as fields in core CSRs in IEEE-1394. Whenever a proper request transaction is received and executed on such a CSR the new value must take effect immediately. This mechanism allows "intelligent" nodes to set operating parameters at other nodes on the bus, for example to optimize performance. In case of a software implementation of these CSRs the CPU must ensure that its local 1394 hardware (link, phy, etc.) behaves accordingly.

4 The PDI1394L11 AVLink chip

The device contains the following major blocks:

- A link layer that interfaces directly to the ’de-facto’ standard PHY interface. It is fully isochronous capable and can operate up to 400 Mbit/s.
- An asynchronous transmitter/receiver, each with a separate fifo (queue) for request and response packets. The transmitter can automatically retransmit busied packets up to a retry limit and detects transaction time-outs. The receiver can match incoming response IDs to an outstanding request ID and filter unsolicited responses or simply receive anything. Received packet are stored in a queue but not processed.
- An isochronous transmitter and receiver which implement a generalized version of the packing protocols for MPEG-2 and DVC. All features required for these data types are present in hardware but can be controlled by programming various parameters. As a result the use of this chip is much more general than just these two data types. It is very likely that it can also be used for many future data types (yet to be standardized in IEC-61883).
- A packet buffer (4 KByte for isochronous data and 256 bytes for each of the 4 asynchronous queues) is implemented to store incoming and outgoing data. Consequently there is a limit to the packet size that is supported for asynchronous traffic and a restriction to a single isochronous stream at a time (either transmitting or receiving).
- An 8 bit CPU interface that supports 8051 type microcontrollers and others is used to access the receive and transmit queues of the asynchronous receiver and transmitter as well as all internal control and status registers in the chip. The CPU does not have to operate at a clock that is synchronous to any internal clock of the AVLink chip.
- An 8 bit AV interface is used to input/output AV data from/to the host application. This interface works autonomously and independent from the CPU interface at a maximum rate of 24 Mbyte/s. The clock rate of the AV interface is defined by an external clock input (from the application).

The AVLink implements all isochronous formatting features required for the data types defined in IEC61883 so far. As for asynchronous transaction handling, only part of the transaction layer (the sequencing mechanism and basic packet handling, such as retransmissions, transaction label matching, time-out detection) is implemented in the chip. The AVLink does not execute transactions but merely passes the packets on to the application for processing (via packet queues). There are no CSRs implemented in the chip, neither IEEE-1394 defined register nor those required by IEC-61883. These should all be implemented in software on the host CPU. Several of the control registers of the chip contain fields that are theoretically part of a CSR but this relation must be enforced in software by keeping the values in these control registers in sync with those in the software emulated CSRs.

The various interfaces and programming models are explained in more detail in the following chapters.

5 PDI1394L11 link layer

The link layer implementation has only one external interface (to the PHY). This interface is described (informative) in Annex J of the IEEE 1394 specification and is compatible with all existing PHY implementations. In addition to the PHY interface there is a cycle clock in (CYCLEIN, pin 30) and cycle clock out (CYCLEOUT, pin 33) signal to/from the link layer which can be used to synchronize the 8 KHz isochronous cycle clock and the application.

5.1 Control and status register of the link layer

There are many controllable features in the link layer, some of which control just the link or even the physical layer but others have an impact on other parts of the chip as well. Some of the fields must be coupled directly to fields in CSRs in the 1394 architecture. Below is a list of all adjustable parameters. Unless explicitly stated otherwise the register contents are not affected by bus resets. See chapter 8 for a register map of the 2nd pass AVLink chip design.

BUS ID: Bus identification number (10 bits), part of 16 bit node ID. This value is inserted in the ‘sourceID’ field of transmitted asynchronous packets and is also used to filter incoming packets. After power-on reset and bus reset this field always reverts to the initial value of 0x3FF (which is required by IEEE-1394). This value is part of the NODE_IDS core CSR in the initial register space of the 1394 architecture. Typically the CPU would write to this field only to execute a write transaction from a bridge manager node (which has algorithms to assign numbers to individual busses).

Node ID: The physical node address (6 bits) of the local node on the local bus. It is used similarly to **BUS ID**, but is also inserted in CIP header of isochronous AV packets. This value will be updated automatically after power-on reset or bus reset with the (new) value determined by the PHY during the automatic bus configuration and self identification phase. This value is part of the NODE_IDS core CSR in the initial register space of the 1394 architecture.

idvalid: Indicates to the link that the values in **BUS ID** and **Node ID** are valid and should be used as advertised. Immediately following a reset (power-on or bus) this bit is cleared, but when the **Node ID** is updated automatically after self identification is complete this bit is set. Normally there is no need for the CPU to write to this field. If this field remains cleared after 20 μs have elapsed since the self ID completed then the automatic update may have failed for some reason. The CPU can request an update at any time by issuing a read command for PHY register 0.

rcvselfid: Enable reception and storage of self-ID and PHY configuration packets. These packets will then be stored in the request queue with a special synthesized packet header. Normally this is required by any node that wants to do isochronous transmissions because an analysis of the self-ID packets is the only way to find out the node ID of the isochronous resource manager where bandwidth and a channel number can be obtained. For that reason the initial value (after power-on reset) is 1.

bsyctrl: Busy acknowledge Control (3 bits). This value controls how the link layer responds to incoming asynchronous packets.

000 = use protocol requested by received packet (either dual phase or single phase)
001 = send a ’busy A’ (testing/diagnostics)
010 = send a ’busy B’ (testing/diagnostics)
011 = use single phase retry protocol
100 = use protocol requested in packet, always send a busy ack (for all packets)
101 = busy A all incoming packets
110 = busy B all incoming packets
111 = use single phase retry protocol, always send a busy ack

Values 0 and 3 are normal operating settings, the others can be used for testing and diagnostics. The initial value after power-on reset is 0.

txenable: This control bit enables the link layer transmitter operation (when set to 1). The initial value after a power-on reset is 1.

rxenable: This control bit enables the link layer receiver operation (when set to 1). The initial value after a power-on reset is 1.

rsttx: This control bit resets the link layer transmitter (when 1). After a power-on reset this bit will be set and automatically cleared after a few clock cycles. When set to 1 by the CPU this bit will not auto-clear.

rstrx: This control bit resets the link layer receiver (when 1). After a power-on reset this bit will be set and automatically cleared after a few clock cycles. When set to 1 by the CPU this bit will not auto-clear.

strictisoch: When set (1) isochronous packets are only accepted when the link layer is in isochronous mode (between cycle start packet and first subaction gap). When clear (0) isochronous packets can be accepted any time. Initial value after power-on reset is 0.

cymaster: This control enables the link layer to operate as cycle master. The cycle timer register must be enabled (**cytmren** = 1), cycle time-out status must be off (**cytmout** = 0) and the node must be root (**root** = 1) otherwise the link will not operate as cycle master. Initial value after power-on reset is 0. This value corresponds to the *cmstr* field of the STATE_CLEAR core CSR of the IEEE-1394 bus architecture.

cysource: Selects external 8 KHz cycle clock (when 1, input pin *cyclein*) or internal cycle clock (when 0) generated from 49.152 MHz input clock from PHY (input pin *sclk*). Initial value after power-on reset is 0.

cytmren: This control bit enables (1) or disables (0) operation of the cycle timer register. This bit must be 1 if the node is involved in isochronous data traffic in any way. Initial value after power-on reset is 0.

CYCTM: Cycle timer register (32 bits). Normally this register does not need to be written, since it is updated periodically with the value received in cycle start packets (except when the node is acting as cycle master). This register consists of 3 fields: **cycle_offset** (12 bits), **cycle_number** (13 bits) and **cycle_seconds** (7 bits).

wrphy: By writing a 1 to this control bit a request is made to the link to write the contents of **phyrgdata** to the PHY register with address **phyrgad**. If all 3 fields are in one control register than they can be written simultaneously to request a write to the PHY. The **wrphy**

field should never be set to 0 and may not be written when either **wrphy** = 1 or **rdphy** = 1. When the request is completed the **wrphy** bit will automatically be cleared.

rdphy: By writing a 1 to this control bit a request is made to the link to read the contents of the PHY register with address **phyrgad**. If both fields are in the same control register than they can be written simultaneously to request a read from the PHY. The **rdphy** field should never be set to 0 and may not be written when either **wrphy** = 1 or **rdphy** = 1. When the request is completed and a result returned by the PHY the **rdphy** bit will automatically be cleared. The last register contents received from the PHY are always available in the (status) fields **phyrxdata** and **phyrxad**. The AVLink will automatically generate a read command for PHY register 0 upon detection of the first subaction gap following a bus reset or power-on reset (to update **Node ID** and **root** values).

phyrgad: PHY register address (4 bits). This is the address of the PHY register to access for read (when **rdphy** = 1) or write (when **wrphy** = 1). See [1], Annex J.4 for a definition of the PHY register map.

phyrgdata: PHY register data (8 bits). This is the data to write to the PHY register addressed by **phyrgad** when **wrphy** = 1.

All control registers can be read back through status registers (same field name). In addition there are several other fields that are read-only. These are:

version_code: A chip version identification code (16 bit). Current value is 0x0002 (constant).

root: This status bit indicates that the node is currently root of the network. It will be updated automatically after each bus reset and power-on reset (like **Node ID** and **idvalid**).

busyflag: This status indicates which busy-ack (A or B) will be returned next by the link layer. It is only meaningful if the dual-phase retry protocol is used for inbound transactions.

atack: Asynchronous transaction acknowledge (4 bits). This is the last received acknowledge value in response to a transmitted asynchronous packet. For diagnostic purposes.

phyrxad: PHY received address (4 bits). The address of the PHY register whose contents were most recently transferred from the PHY to the AVLink, typically in response to a read register request.

phyrxdata: PHY received data (8 bits). The contents of the PHY register indicated by **phyrxad** at the time it was transferred to the AVLink (note that this was a snapshot of the register contents and its value may have changed since then).

In addition to these functional controls there are also interrupt/event status and control bits. All signals below are event indications which means that the (status) bit will be set whenever the corresponding event occurs. The status bit will be cleared (acknowledged) only by writing a 1 to the corresponding acknowledge register, thus allowing each event to be handled and acknowledged separately. Furthermore each of these event indications can be used as an interrupt source by setting a corresponding mask bit to 1 (interrupt enable bits).

cmdrst: Indicates that the link has received a write request to the RESET_START core CSR. Such a write request represents a so called *command reset* which takes precedence over any other queued or pending transactions and must be handled immediately. For that reason the link layer intercepts this particular transaction and translates it into a **cmdrst** event indication, rather than queueing the packet. For information on how to react to command resets, refer to the IEEE-1394 standard.

fairgap: The PHY reported the detection of a subaction gap on the cable.

arbgap: The PHY reported the detection of a arbitration reset gap on the cable.

phyint: The PHY reported a time-out. An example is the configuration time-out during tree-identify phase after a bus reset. If a cable loop is present the standard tree identify algorithm hangs and the bus cannot operate. The PHY detects this by a time-out mechanism and reports it to the AVLink which then raises the **phyint** event indication.

phyrrx: The AVLink has received the contents of a PHY register, typically in response to a read request (**rdphy**) issued earlier.

phyrst: The PHY reported the detection of a bus reset in progress on the cable. This event is extremely important and is probably best handled at interrupt level. It also causes the asynchronous buffers to be emptied and the receiver buffers to be locked for read access. Several asynchronous registers will revert to initial values as specified in IEEE 1394 and software implemented CSRs should revert to initial values as well. The list of tasks to perform after a bus reset is extensive and out of the scope of this document. See [5] for more detailed information.

txrdy: The transmitter in the link layer reported being ready and idle. For diagnostic purposes.

rxdata: The receiver in the link layer wrote some data in one of the receiver fifos. For diagnostic purposes only.

itbadfmt: The link layer transmitter detected incorrectly formatted data at its isochronous transmitter input. This should never occur and is included for completeness sake.

atbadfmt: The link layer transmitter detected incorrectly formatted data at its asynchronous transmitter input, typically the result of failing to write the last quadlet of each packet to a separate location from the other quadlets (see asynchronous interface).

snt_rej: An incoming packet was busy-acked (rejected) by the receiver.

hdrerr: A header CRC error was detected in an incoming packet. The packet has been discarded conform IEEE-1394 standard.

tcerr: An invalid transaction code was presented to the asynchronous transmitter.

cytmout: Isochronous cycle lasted too long (isochronous ‘overload’ or violation). If the node is cycle master then it will no longer send cycle start packets to allow asynchronous traffic to resolve the problem. Cycle master functionality can be restored by the following procedure: (1) set **cymaster** to 0. (2) acknowledge **cytmout** event by writing a 1 to it. (3) set **cymaster** back to 1. Do not do this until the problem that caused the violation has been fixed, otherwise there will just be another cycle time-out.

cysec: Cycle seconds field increment. This event occurs once every second, when the cycle seconds field in the **CYCTM** increments.

cystart: Cycle started. This indicates that a cycle start packet has just been sent (if cycle master) or received (if not cycle master). As a result the link layer has entered isochronous mode which will last, or lasted, until the detection of the first subsequent subaction gap (**fairgap** event). For a proper interpretation of these events the CPU should handle them in ’real-time’, i.e. fast enough to see them occur one at a time, and it should clear (acknowledge) this event indication before the next cycle starts.

cydone: Cycle done. This indicates that an isochronous cycle has ended (due to detection of a subaction gap). For proper interpretation the CPU should clear (acknowledge) this event indication before the next cycle starts (**cystart**).

cypend: Cycle pending. Indicates that it is time to start a new isochronous cycle (according to **CYCTM** value), but a cycle start packet has not yet been sent or received.

cylost: Cycle lost. Indicates that an entire isochronous cycle was lost. This happens when it is time for another cycle start packet while **cypend** is still 1. Possible causes are a bus reset in progress, or a missing cycle start packet due to transmission error or failing cycle master. If an isochronous transmitter in the AVLink is active it will discard an entire payload as a result of this indication.

Finally there is one more global chip control register which is not part of any specific module. It contains the following control bits:

txmode: Selects transmit mode (1) or receive mode (0) for isochronous part of AV layer. This also controls the direction (in/out) of the AV interface (see chapter 6).

easytx/rx: Enables generation of external interrupt by asynchronous interface when set. Functions as a master interrupt enable for the entire asynchronous interface.

eitxint: Enables generation of external interrupt by isochronous transmitter when set. Functions as a master interrupt enable for the entire isochronous receiver.

eirxint: Enables generation of external interrupt by isochronous receiver when set. Functions as a master interrupt enable for the entire isochronous receiver.

elnkphyint: Enables generation of external interrupt by link layer when set. Functions as a master interrupt enable for the entire link layer.

status bits:

asyitx/rx: External interrupt is caused by enabled interrupt in asynchronous interface.

itxint: External interrupt is caused by enabled interrupt in isochronous transmitter.

irxint: External interrupt is caused by enabled interrupt in isochronous receiver.

lnkphyint: External interrupt is caused by enabled interrupt in link layer.

6 PDI1394L11 isochronous data transfers

6.1 AV interfaces

Internally both the IEC-61883 transmitter and IEC-61883 receiver have their own AV interface. To save pins (and because the current chip does not handle simultaneous data transmit and receive) the 2 AV interfaces are mapped to the same set of pins. Most pins of the AV interface are shared (those with the same name) and switch direction (input/output) depending on the mode setting in the chip. Some other pins are specific for the transmitter or receiver and are not shared.

6.1.1 The transmitter AV interface

The transmitter uses an 8 bit data input (*avdata*), a data valid indication (*avvalid*), a start of packet synchronization signal (*avsync*) and an end of packet synchronization signal (*avendpck*). The entire interface runs at a clock generated by the application (*avclk* input). In addition to these signals there is one more input for frame synchronization (*avfsyncin*) which does not have to be clock synchronous to any clock (including *avclk*) and can be used to direct the transmitter to send a *syt* stamp if this feature is enabled. Below is a timing diagram showing an example of these signals. **Note: *avsync* must be qualified by *avvalid*. Do not assert *avsync* without *avvalid* asserted.**

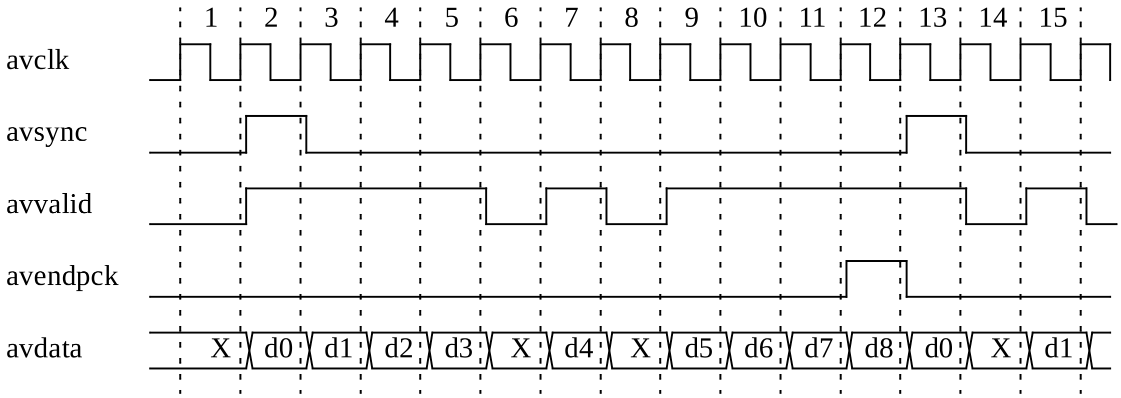


Figure 2: Example of signalling on the transmitter AV interface.

Signal *avfsyncin* is not shown here because it has no relation to these signals. The application packet size in figure 2 is 9 bytes (d0..d8). Signal changes are always based on the rising edge of *avclk*. The chip samples inputs at the rising edge.

The start of a packet is indicated by simultaneously raising *avsync* and *avvalid* during one clock cycle (cycles 2 and 13 in figure 2). At the same time the first data byte (d0) must be present at the *avdata* input. During the subsequent clock cycles the transmitter samples *avvalid* and accepts data on the *avdata* input whenever *avvalid* is asserted. Hence data does not necessarily have to be input in consecutive clock cycles. For example in figure 2 there is no valid data during clock cycles 6, 8 and 14 as indicated by *avvalid*. The value at the *avdata* input during these cycles is ignored by the transmitter.

The end of a packet is indicated by simultaneously raising *avendpck* and *avvalid* when the last data byte is present on *avdata*. The use of *avendpck* is required for application packets whose length is not a multiple of

4 and optional for packets that are a multiple of 4 bytes in size. In particular for MPEG-2 TS and DVC-SD/HD this signal is optional and can be tied to zero.

For application packets that are not quadlet aligned (not a multiple of 4 bytes in size) the transmitter will automatically pad bytes (undefined value) to make them quadlet aligned. The padding can be removed by the receiver but since no information about byte padding is carried in the stream the CPU on the receiving side must instruct the receiver to do so (e.g. based on the data format from the *fmt* field).

As can be seen in figure 2 the next application packet can start immediately after the end of the previous one (cycles 12 and 13). This is called back-to-back packet input. There may also be an arbitrary period (integer number of clock cycles) of idleness (*avvalid* = 0) between two consecutive application packets.

The maximum frequency for avclk that is currently supported is 24 MHz with one exception: for back-to-back input of application packets that require 3 bytes of padding (packet size = 4N+1) the maximum supported frequency is 16 MHz. Maximum data rates are therefore set at approx. 24 MByte/s.

The *avfsyncin* input can be used to send a *syt* stamp if this feature has been enabled (by programming the internal control registers of the AVLink chip. In that case the *syt* field is normally filled with all 1’s and the transmitter will sample the *avfsyncin* pin to detect a rising edge on this signal. To guarantee that the transmitter will see the rising edge the pulse width should not be less than 42 ns. The value for the *syt* field is computed by sampling the lower 16 bits of the Cycle Timer Register of the local LINK and adding 3 to the value of the sampled (4 bit) cycle_number (making the stamp point 375 µs into the future). The *syt* value is then sent in the CIP header of the next transmitted packet (only once). The transmitter-receiver combination can handle *avfsync* pulses up to 2 KHz.

6.1.2 The receiver AV interface

The receiver uses an 8 bit data output (*avdata*), a data valid indication (*avvalid*), a start of packet synchronization signal (*avsync*) and a status indication (*averr*). The entire interface runs at a clock generated by the application (*avclk* input). In addition to these signals there is one more output for frame synchronization (*avfsyncout*) which is synchronous to the internal 24.576 MHz clock and is controlled by incoming *syt* stamps, if this feature is enabled. Below is a timing diagram showing an example of these signals.

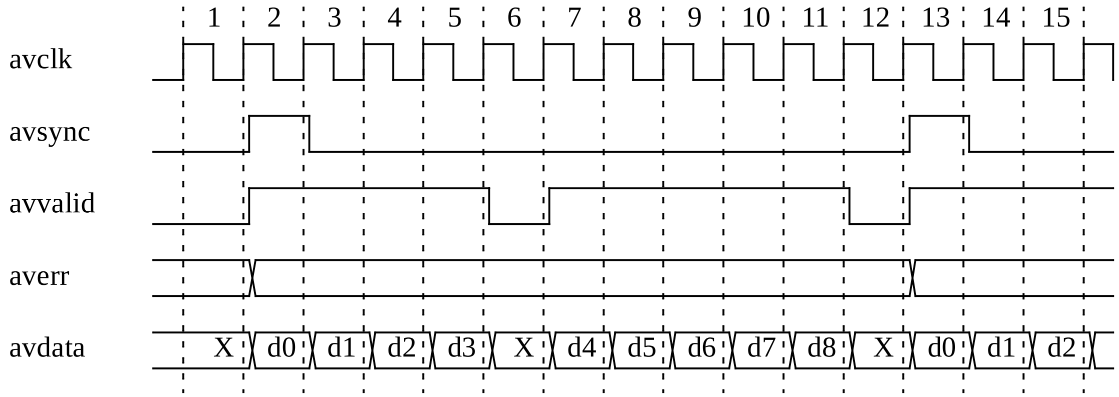


Figure 3: Example of signalling on the receiver AV interface.

All signals except *avclk* are outputs. Similar to the transmitter the start of a new packet is indicated by the receiver by raising *avsync* and *avvalid* simultaneously during one clock cycle. At the same time the first data byte is present at the *avdata* output. The data will be delivered during the subsequent clock cycles, typically without interruption. On rare occasions (with very high data rates) the receiver may be forced to negate *avvalid* for one or two clock cycles because it could not keep up the data rate (example: clock cycle 6 in figure 3).

The *averr* output contains a 2 bit status indication of the current packet. This signal remains constant from the first up to and including the last byte of the packet. Bit 0 indicates that part of this application packet was received in a bus packet with a data CRC error. Consequently the packet contents are unreliable. Bit 1 indicates that this application packet is the first completely received packet after a sequence error had been detected (usually after a missing bus packet, e.g. after a bus reset). If this bit is set there is nothing wrong with the current packet but some data was lost before this packet. Since some applications may have trouble processing data with bit errors (MPEG decoders) the AVLink chip offers the possibility of having these packets removed rather than delivered to the application.

The receiver is capable of back-to-back packet delivery. Timing of delivery is based on arrival of the entire packet (possible spread across many bus packets) and time stamps (if used). An application packet becomes ready for delivery if all its data blocks have been received completely and the CRC of the bus packet containing the last data block has been checked and its stamp (if any) expires. If the AV interface is idle and a new application packet is ready for delivery it is fetched from packet memory and output on the AV interface.

The maximum frequency for *avclk* supported in all cases is 24 MHz, allowing data rates of up to 24 MByte/s. For high rates the receiver may have to negate *avvalid* occasionally thereby reducing the effective data rate, especially for byte padded and/or very small (≤ 4 bytes) application packets.

Processing of *syt* values can be enabled or disabled by software. If enabled the receiver will treat the lower 16 bits of the *fdf* field as *syt* stamps and whenever a valid value (not all 1's) is received it will be used to generate a correctly timed pulse on the *avfsyncout* output. This pulse will occur when the received *syt* value matches the lower 16 bits of the Cycle Timer Register in the local LINK. The duration of the (active high) pulse is 3 cycles of the internal 24.576 MHz clock.

6.2 Control and status registers for the isochronous transmitter

The AV transmitter has several quadlet wide control and status registers. This section discusses all fields in these registers and their meaning/use. See also chapter 8.

rst_itx: When this bit is set the AV transmitter logic is reset, but the control registers keep their old value. Use this to recover from serious errors such as buffer overflow, or to clear all counters and memory if a new link transmission needs to be set up. Default value at power reset is 1 (active). This bit will not clear by itself.

en_fs: Enables (1) or disables (0) processing of *avfsyncin* pulses and the creation and transmission of *syt* stamps in the 2nd CIP header quadlet.

en_itx: Enables (1) or disables (0) transmitter operation. The transmitter has two interfaces: one to the application (AV interface) and one to the link layer. Both are affected by this control bit but not until they have become inactive. This means that if any of these interfaces is busy transferring data that operation will be completed before being disabled. This prevents partial packets being stored/transmitted. Consequently a transmitter shutdown can be delayed by as much as 125 µs on the link side, and the duration of an application packet on the application side. This must be taken into account by software drivers.

pm: Packing mode (2 bit). Indicates how bus packets are constructed from data blocks stored in memory, as follows:

00: The transmitter will put as many data blocks as possible in each payload. The maximum is limited only by the **maxbl** parameter which the application must set. If no data blocks are available then only a CIP header will be sent. There are no alignment restrictions.

01: The transmitter will put a fixed number of data blocks (indicated by **maxbl** parameter) in each payload. If that many data blocks are not available when the next packet must be transmitted then the payload will contain just a CIP header.

10: The transmitter will use the MPEG-2 alignment and packing rules. If the **maxbl** parameter indicates that the maximum payload size is less than a source packet then the rules for packing mode 1 (**pm** = 01) apply with additional alignment restrictions as explained for MPEG earlier (see section 2.2.1), otherwise the payload will contain an integer number of source packets (k * 2^{fn} data blocks). The payload size will never exceed **maxbl** data blocks. This mode works for all settings of *fn*, and the alignment criteria change accordingly. A change in **maxbl** might not take effect immediately (see below).

11: The transmitter will never put any data blocks in a payload. Every payload contains just a CIP header with proper field values. This can be used to synchronize applications (*syt* stamps) before actually commencing data transmission.

Note: The value of **pm** may not be changed while the transmitter is enabled.

maxbl: Maximum payload size in data blocks (8 bits). Depending on **pm** this parameter defines the exact or maximum number of data blocks that the transmitter may/will send in a single isochronous payload. The value of **maxbl** may not be changed while the transmitter is enabled, except in packing mode 2 (**pm** = 10, MPEG-2 mode). In case of packing mode 2, a change in **maxbl** takes effect when, at the end of a packet transmission, the next data block to be transmitted is in accordance with the alignment criteria corresponding to the new **maxbl** value (the worst case delay for this is 8 isochronous cycles, or approximately 1 msec.).

trdel: Transport delay (12 bits). This parameter specifies the offset to be added to the sampled 25 bits of the Cycle Timer Register (CYCTM) for the creation of a packet delivery time stamp. The 12 bit **trdel** value is extended to the right with 8 zeroes and to the left with 12 zeroes and then added in the format of the CYCTM. Effectively this means that the lower 4 bits of **trdel** add to the upper 4 bits of the cycle_offset (modulo 3072) and the remaining bits add to the cycle_number (modulo 8000), allowing the transport delay to be set from 0 to 32 msec. in increments of approximately 8 µs. This value is used only if **sph** = 1.

sph: Source Packet Header. When set the transmitter will attach a packet delivery time stamp to every application packet, allowing packet timing reconstruction at the receiving node.

qpc: Quadlet Padding Count (3 bits). The number of (zero-filled) quadlets of padding that are inserted into the data stream at appropriate times to create the source packets.

fn: Fraction Numbering (2 bits). Encodes the number of data blocks (fractions) into which each source packet is divided: number of data blocks = 2^{fn}.

dbs: Data Block Size (8 bits). The size of the resulting data blocks in quadlets. A value of 0 actually means a data block size of 256 quadlets.

fmt: Format (6 bits). This field is inserted in the second CIP header quadlet *fmt* field. It is not interpreted by the transmitter in any way.

fdf/syt: Format Dependent Flags (24 bits). This field is inserted in the second CIP header quadlet *fdf/syt* field and is not interpreted by the transmitter. However, if **en_fs** = 1 then the lower 16 bits of **fdf/syt** are ignored and the lower 16 bits of the second CIP header quadlet are either all 1's (if no *syt* available for transmission) or a valid *syt* value.

tag: Isochronous Tag value (2 bits). The transmitter will insert this value in the 2 bit *tag* field of the 1394 link level isochronous packet header.

chan: Isochronous channel (6 bits). The transmitter will insert this value in the channel number field of the 1394 link level isochronous packet header.

spd: Isochronous speed (2 bits). The transmitter will request the link layer to transmit packets at the speed indicated in this field:

00: speed S100 (98.304 MBit/s)

01: speed S200 (196.608 MBit/s)

10: speed S400 (393.216 MBit/s)

11: reserved

sync: Isochronous sync value (4 bits). The transmitter will insert this value in the *sy* field of the 1394 link level isochronous packet header.

In addition to these control registers there are several (programmable) interrupt sources or event indicators, mostly for diagnostics and errors (which normally should not occur). Each source can be separately enabled to generate an interrupt and each source can be separately acknowledged. All control registers can be read back from the AVLink chip. Furthermore some status about the memory bank is provided.

The event indications / interrupt sources are:

itxempty: Isochronous memory bank did become empty (no data to transmit).

itxfull: Isochronous memory bank did fill up completely (possible overflow!).

discard: Payload was discarded instead of transmitted because of a lost cycle (usually after bus reset).

inperr: Input data discarded. Input data did not correspond to programmed source packet size and data had to be discarded to fix it. Represents an application error. Not the AVLink's fault. Problem will self-correct once application resumes proper formatting of input data.

dbcerr: Data block sequence counter misalignment detected and repaired. Transmitter verifies that each first data block of a source packet has a sequence number with the lowest **fn** bits all 0. If it detects an error the problem is corrected by enforcing this. Diagnostic purpose: this should never go wrong under normal operating conditions.

trmbp: A packet transmission has been completed, or a payload has been discarded (when the transmitter is enabled this event will occur every cycle!). Diagnostic purpose.

trmsyt: A syt stamp has been transmitted in a CIP header.

Each of these events has a corresponding interrupt enable bit whose name equals the event name prefixed by ’**e**’.

The memory status indications are (for diagnostic purposes):

itxmf: Isochronous transmitter buffer memory is currently full.

itxmaf: Isochronous transmitter buffer memory has precisely one quadlet available.

itxm5av: Isochronous transmitter buffer memory has at least 5 quadlets available.

itxme: Isochronous transmitter buffer memory is empty.

6.3 Parameter settings for isochronous transmissions

The large number of programmable parameters make this chip very versatile. It also makes it more difficult to use the chip properly. For data types defined in IEC-61883 the settings are easy: just take the values from the standard specification and write them to the corresponding register field. The remainder of this section explains in more detail the intricacies of creating a set of working parameters for any specific data type.

6.3.1 Finding a set of working parameters

Assume an application packet size of N bytes. The following steps can be taken to come up with a valid set of parameters:

1 Round up N to multiple of 4 bytes to align to quadlets. This requires 0..3 bytes of padding. The information on how many bytes need to be added is lost in the transmission, but could be carried in the *fmt* and *fdf/syt* field (requires standardization by IEC). The case with 0 bytes padding always works.

2 Divide the result by 4 to get the application packet size in quadlets, called NQ.

3 If constant transport delay is wanted (low jitter) then set **sph** = 1 else set **sph** = 0. Adding a stamp will increase the data size by 4 bytes for every application packet.

4 Select a suitable setting for **fn**, **dbs** and **qpc** such that $2^{\mathbf{fn}} * \mathbf{dbs}$ **- qpc - sph** = NQ where **qpc** must be in the range 0..**min**($2^{\mathbf{fn}}$, **dbs**)-1, **dbs** is in the range 1..256 and **fn** is in the range 0..3. In this formula $2^{\mathbf{fn}} *$ **dbs** is the source packet size in quadlets, **qpc** is the number of dummy quadlets added as quadlet padding (0..7) and **sph** is the number of quadlets added for stamps (0 or 1). This clearly shows the 1:1 relation between application packets and source packets.

5 Determine a packing mode for this data (0=variable, 1=fixed, 2=mixed). If the bit rate can change midstream then packing mode 2 is the only option, otherwise all 3 are possible. Mode 0 needs less buffer space due to a reduced payload size granularity.

6 Determine the value of **maxbl**. Starting with the (constant) bit rate from the application, calculate the number of application packets per cycle:

$$\mathrm{npc} = \frac{\mathrm{abr} \times 125 \times 10^{-6}}{\mathrm{N} \times 8}$$

where **npc** is the number of application packets per cycle and **abr** is the application bit rate. Since each application packet is mapped into a source packet this is also the number of source packets per cycle.

The number of data blocks stored in memory per cycle **ndbc** now becomes:

$$\mathrm{ndbc} = \mathrm{npc} \times 2^{\mathrm{fn}}$$

The value of **maxbl** must be set to at least this value to prevent memory overflows in the transmitter. If special restrictions apply (such as for packing modes 1 and 2 where payload sizes are restricted) then the value should be such that on average at least **ndbc** data blocks can be transmitted.
Given the values of **pm**, **maxbl, ndbc** and **dbs** the largest payload size can be computed. An upper bound is easily found to be **maxbl** ∗ **dbs** ∗ 4 + 8 bytes. The 8 bytes are for the CIP header. At the link layer another 3 quadlets are added for isochronous header and CRCs. At the physical layer some more overhead is added for arbitration, etc. The resulting total bandwidth is what needs to be allocated before any isochronous transmission may take place.

7 Find a suitable transport delay (only if **sph** = 1). The range of values is defined by the application data rate and buffer size: delay_max = buffer_size / abr. In the current AVLink chip design the buffer size is 32768 bits (4 KByte). This simple expression is valid only under the assumption that the appl_bit_rate is constant (i.e. the stream is not of a bursty nature). If packet arrival is not at a constant rate (such as after program selection from an MPEG stream) then this effect should be taken into account and computation of a valid maximum delay is much more complex and out of the scope of this document. The absolute minimum delay value (delay_min) is about 3 isochronous cycles (375 μs) if an exact integer number of application packets is transported in every cycle, or 4 isochronous cycles (500 μs) otherwise, due to the transport granularity. A practical minimal delay however depends on the bit rate, **pm** and **maxbl** setting and could be far higher than 3 or 4 cycles.

6.3.2 Example

Below is an example for some arbitrarily chosen (non-existent) data type.

Consider a data type that uses packet of 63 bytes and operates at a constant bit rate of 26 MBit/s. The data is time critical so stamp based (low jitter) delivery of packets is required.

1 One byte of padding must be added to create a data length that is divisible by 4.

2 The resulting packet size NQ is 16 quadlets.

3 Constant delay is required, so **sph** = 1.

4 A possible setting is **fn** = 0, **qpc** = 0, **dbs** = 17; another one is **fn** = 1, **qpc** = 1, **dbs** = 9. There are many options. For this example, assume the second one.

5 Arbitrary choice: **pm** = 1 (fixed size bus packets).

6 **npc** = (26 x 125) / (63 x 8) = 6.45 appl. packets/cycle. => **maxbl** ≥ 13.
Since bus packet size is fixed in this case the payload size is 13 * 9 * 4 + 8 = 476 bytes and each bus packet will carry 13 data blocks, or 6.5 application packets. If **maxbl** is increased to 14 each bus packet will carry 7 data blocks and the system will be somewhat less sensitive to missing bus packets or bit errors on the cable since no application packet is spread out over more than one bus packet.

7 To compute a transport delay, approximately 0.5 KByte is needed for granularity, etc. (buffering used to handle the packing and formatting and to save data until the next cycle in case there was just not enough data available to do a (fixed size) transmission. Consequently 3.5 KByte (28 Kbit) is left for jitter and transport delay buffer: 28E3/26E6 = 1.1 msec => 8 cycles. The value of **trdel** can be set from 4 to 8 cycles.

An example where the lowest transport delay must be set much higher is low bit rate MPEG-2. For example at 2 Mbit/s MPEG-2 transport packets arrive every 752 µs, or once every 6.02 cycles. The stamp is created when the first byte enters the chip at the AV interface, the last byte enters the chip almost 752 µs later. The first data block(s) may already have been transmitted to the receiver before that time (including the stamp). The last data block is transmitted 6 or more cycles after the first data block in that case. IEC-61883 specifies that the stamp may not expire between the moment of its creation to the moment when the last bit of the bus packet containing the last data block appears on the cable and that it is the transmitter's responsibility to guarantee this. Obviously the **trdel** parameter should be set to some value greater than 6 in this example, preferably a few cycles extra as a safety margin.

6.4 Operating the isochronous transmitter

The previous sections explained the various controls and parameters and how to obtain a working set of value. There are some additional rules that must be taken into account to use this transmitter properly on a 1394 bus.

- None of the settings may be changed while the transmitter is active (enabled). The only exception to this rule is that in packing mode 2 the value of **maxbl** may be changed at any time. Changing any setting other than this will cause incorrect behavior of the AVLink and possible protocol violations.
- The packing parameters **dbs**, **fn**, **qpc**, **sph** may not be changed without subsequently resetting the transmitter to clear the buffer of any stored data (unless it can be otherwise guaranteed that the buffer is empty). These same holds for an increase of parameter **trdel** in case **sph** = 1 (a decrease is safe) because transmitted stamps must always form a monotonically increasing series of values. Failing to empty the buffer after a change in any of these parameters will cause incorrect behavior of the AVLink.
- The packing mode **pm** may be changed only when the transmitter is disabled. It is not necessary to clear the buffer after a change in pm since **pm** does not affect the format of source packets. The same applies to **maxbl**, unless **pm** = 2 in which case **maxbl** may be changed any time.
- The other parameters (**en_fs**, **fmt**, **fdf/syt**) may be changed at any time.
- To disable the transmitter, program **en_itx** to 0 and wait long enough to guarantee that all packet I/O has stopped. To reset the transmitter and clear the buffer program **rst_itx** to 1 and then back to 0.
- The 1394 link level parameters (**spd**, **chan**, **sync**) may be changed at any time. The tag value **tag** should be ’01’ for IEC-61883 compliance, but could have any value for AVLink.

As a consequence of the above rules the only good way of setting up a new transmission link is the following series of steps: determine the new parameter values, disable the transmitter, then reset it, then install the new parameters and finally enable the transmitter.

6.5 Control and status registers for the isochronous receiver

The receiver is also generic in design. It takes an incoming data stream (payloads) and reconstructs the original application packets with timing (if required). It has fewer control parameters because it extracts all the information it needs to process the data from the received CIP headers. Nevertheless there are a number of control registers. See also chapter 8.

rst_irx: When this bit is set the AV receiver logic is reset, but the control registers keep their old value. Use this to recover from serious errors such as buffer overflow, or to clear all counters and memory if a new data transfer is being set up. Default value at power reset is 1 (active). This bit will not clear by itself.

en_fs: Enables (1) or disables (0) processing of received *syt* stamps in the 2nd CIP header quadlet and corresponding generation of *avfsyncout* pulses.

en_irx: Enables (1) or disables (0) receiver operation. The receiver has two interfaces: one to the application (AV interface) and one to the link layer. Both are affected by this control bit but not until they have become inactive. This means that if any of these interfaces is busy transferring data that operation will be completed before being disabled. This prevents partial packets being stored/transmitted. Consequently a receiver shutdown can be delayed by as much as 100 µs on the link side, and the duration of an application packet on the application side. This must be taken into account by software drivers.

bpad: Byte padding (2 bits). The receiver will remove the last **bpad** bytes from every application packet. This counter-acts the effects of byte padding performed by a transmitter. Since byte padding counts are not carried across the link the (any) receiver would normally recover application packets with a length that is a multiple of 4 bytes, unless instructed otherwise by the application who must obtain this information in some other way.

rmvuap: Remove unreliable application packets. When set, this control bit instructs the receiver to discard application packets that would otherwise get delivered to the application with a status indication of CRC error (meaning that at least one data block for this application packet was received in a bus packet with a data CRC error. The data will still appear on the AV interface with proper timing (*avdata*) but the *avvalid* and *avsync* signals will not be raised.

chan: The receiver will filter incoming packets such that only isochronous packets sent on the channel indicated by **chan** and proper tag value (**tag**) are processed.

tag: The receiver will filter incoming packets such that only isochronous packets with the proper tag field value (indicated by **tag**) and proper channel value (**chan**) are processed.

In addition to these controls there are several event indications (status) which are all independently programmable as an interrupt sources and which can all be independently acknowledged as interrupt. All above control registers can be read back from the chip, as well as several receiver status indications.

The event indications / interrupt sources are:

irxfull: Isochronous memory bank did become full (possible overflow!).

irxempty: Isochronous memory bank did become empty (no incoming data).

fsync: A pulse was generated on the *avfsyncout* pin as a result of a *syt* expiration.

seqerr: A sequence error was detected (DBC continuity error) typically the result of a missing bus packet after a bus reset.

crcerr: A bus packet was received with a CRC error for the data field. All data in the bus packet is now marked as unreliable but stamps will be processed as if they are OK!

ciptagflt: A CIP header with an unsupported format was received (E and F flags). The packet contents have been discarded.

rcvbp: A bus packet has been received and processed. This is meant for diagnostics only, but can also be used to detect the presence of a transmitter on the selected channel/tag.

sqov: Internal status queue overflow. The current AVLink receiver can store up to 32 source packets, or up to 4 KByte (whichever runs out first). If a 33rd source packet must be stored this overflow will be set. This is a serious error which may not self-repair after a while. Recommended handling is to reset the receiver. This error will not occur for DVC and MPEG since buffer space (4 KByte) will run out before 32 source packets are stored.

All events have an interrupt enable bit whose name equals the event prefixed by '**e**'.

The memory status indications are (for diagnostic purposes):

irxmf: Isochronous receiver buffer memory is currently full.

irxmaf: Isochronous receiver buffer memory has precisely one quadlet available.

irxm5av: Isochronous receiver buffer memory has at least 5 quadlets available.

irxme: Isochronous receiver buffer memory is empty.

Available status values:

spav: Source packet available. Indicates that an entire source packet has been received and stored in memory and that it is now awaiting delivery.

e0, e1, f0, f1, sid, dbs, qpc, fn, sph, fmt, fdf: Values for the corresponding fields in the last received and processed (if any) CIP header.

spd: The speed at which the last processed packet was received:

00: speed S100 (98.304 MBit/s)

01: speed S200 (196.608 MBit/s)

10: speed S400 (393.216 MBit/s)

11: reserved

err: The 4 bit error code (dummy-ack) for the previously processed packet.

sync: The 4 bit contents of the *sy* field of the previously processed packet.

7 Asynchronous packet transceiver

The asynchronous transmitter/receiver each have a separate fifo (queue) for request and response packets. The transmitter can automatically retransmit busied packets up to an adjustable retry limit and detects transaction time-outs. The receiver matches incoming response IDs to the currently outstanding request ID and can be programmed to filter unsolicited responses or simply receive anything. Each transmitted packet results in a confirmation which is stored by the receiver in the corresponding fifo depending on the packet type (*request*/*response*) that was transmitted.

The entire transceiver is operated by writing control and reading status bits of the AVLink chip's registers. These registers can be accessed by a simple protocol using an 8 bit CPU interface (see chapter 9).

7.1 Control and status registers of the asynchronous transmitter

Control registers/bits:

atxrst: Asynchronous transmitter reset bit. As long as this bit is set the transmitter logic in the asynchronous interface is reset and the transmitter queues are being cleared. This bit will be set after a power-on reset and after each bus reset.

maxrc: The maximum retry count value. This value controls the maximum number of times a packet transmission will be retried if the destination node is busy. The default value after a power reset and each bus reset is 0. Note: the transmitter will always give precedence to the other fifo (if it holds a complete packet) before retransmitting a packet. This value corresponds to the retry_limit field in the BUSY_TIMEOUT core CSR of the IEEE-1394 architecture.

tos/**tof**: Split transaction time-out value. This time-out is specified in two fields: an integer and a fractional part. The **tos** field holds the integer part (in units of a second) and the **tof** field holds the fractional part (in units of cycles, 1/8000 of a second). This value determines the maximum amount of time the transmitter will wait for a pending response to a transmitted request packet, before transmitting the next available request packet. The default split time out value after a power reset and each bus reset is 100ms. This value corresponds to the SPLIT_TIMEOUT core CSR of the IEEE-1394 architecture.

tx_rq_next: Quadlet wide register to which the first and middle quadlets of a request packet must be written to store them into the request packet fifo.

tx_rq_last: Quadlet wide register to which the last quadlet of a request packet must be written to store it into the request packet fifo.

tx_rp_next: Quadlet wide register to which the first and middle quadlets of all non request packets must be written to store them into the response packet fifo.

tx_rp_last: Quadlet wide register to which the last quadlet of all non request packets must be written to store it into the request packet fifo.

In addition to these control registers/bits there are several (programmable) interrupt source bits or event indication bits, mostly for diagnostic purposes and error indications. Each source can be separately enabled to generate an interrupt and each source can be separately acknowledged. Also some status registers about the fifos are provided.

The event indication / interrupt source bits are:

treqqwr: Transmitter request queue (fifo) written. This bit gets set if the quadlet that was written to the **tx_rq_next** or **tx_rq_last** register has been transferred to the request fifo. This event can be used to synchronize CPU write accesses with the speed of the AVLink chip's internal logic. This bit is cleared by acknowledging it or by simply writing the next quadlet if any to either the **tx_rq_next** or **tx_rq_last** register.

treqqwrerr: Transmitter request queue (fifo) write (synchronization) error. This bit gets set if the AVLink chip's internal logic could not keep up with the CPU write accesses to the **tx_rq_next** or **tx_rq_last** register.

treqqfull: Transmitter request queue (fifo) full. The request fifo did fill up completely at least once since this event was last acknowledged.

timeout: This bit gets set if the response to a transmitted request packet in case of a split transaction has not been received within the time specified by the split transaction time out value.

rcvdrsp: Received response. This bit gets set if the response to a transmitted request packet in case of a split transaction has been received within the time specified by the split transaction time out value.

trspqwr: Transmitter response queue (fifo) written. This bit gets set if the quadlet that was written to the **tx_rp_next** or **tx_rp_last** register has been transferred to the response fifo. This event can be used to synchronize CPU write accesses with the speed of the AVLink chip's internal logic. This bit is cleared by acknowledging it or by simply writing the next quadlet if any to either the **tx_rp_next** or **tx_rp_last** register.

trspqwrerr: Transmitter response queue (fifo) write (synchronization) error. This bit gets set if the AVLink chip's internal logic could not keep up with the CPU write accesses to the **tx_rp_next** or **tx_rp_last** register.

trspqfull: Transmitter response queue (fifo) full. The response fifo did fill up completely at least once since this event was last acknowledged.

All these events have an interrupt enable bit whose name equals the event prefixed by '**e**'.

The memory status indication bits are (for diagnostic purposes):

treqqidle: Transmitter request queue (fifo) idle. This bit indicates that there is no quadlet in either the **tx_rq_next** or **tx_rq_last** register.

treqqf: Transmitter request queue (fifo) full. The fifo is currently full.

treqqaf: Transmitter request queue (fifo) almost full. The fifo has precisely space for one more quadlet.

treqq5av: Transmitter request queue (fifo) five available. The fifo has at least space for five more quadlets.

treqqe: Transmitter request queue (fifo) empty. The fifo is empty.

trspqidle: Transmitter response queue (fifo) idle. This bit indicates that there is no quadlet in either the **tx_rp_next** or **tx_rp_last** register.

trspqf: Transmitter response queue (fifo) full. The fifo is currently full.

trspqaf: Transmitter response queue (fifo) almost full. The fifo has precisely space for one more quadlet.

trspq5av: Transmitter response queue (fifo) five available. The fifo has at least space for five more quadlets.

trspqe: Transmitter response queue (fifo) empty. The fifo is empty.

7.2 Control and status registers of the asynchronous receiver

Control registers/bits:

arxrst: Asynchronous receiver reset bit. When set, then bit clears the receiver queue and reinitializes the receiver logic in the AV layer. This control bit auto clears at the earliest appropriate moment (when no packet is coming in through the cable) to prevent reception of partial packets. This bit always reads as ’0’.

arxall: Asynchronous receiver receive all bit. If this bit is cleared the receiver will not write unsolicited response into the receiver response queue. If this bit is set the receiver will store all response packets it receives in the response queue.This bit does not affect other packet types, such as request packets. A special tag bit in the last quadlet of the packet can be used to distinguish solicited and unsolicited responses. The default value of the **arxall** bit is zero after a power reset or each bus reset.

rreq: Receiver request register. Register from which the quadlets of a received packet present in the request fifo are read.

rrsp: Receiver response register. Register from which the quadlets of a received packet present in the response fifo are read.

In addition to this control registers/bit there are several (programmable) interrupt source bits or event indication bits, mostly for diagnostic purposes and error indications. Each source can be separately enabled to generate an interrupt and each source can be separately acknowledged. Also some status registers about the fifos are provided.

The event indication bits / interrupt sources bits are:

rreqqqav: Receiver request queue (fifo) quadlet available. This bit indicates that a quadlet of a packet queued in the request fifo is available for reading from the **rreq** register. This bit can be cleared by acknowledgment or by simply reading the quadlet.

rreqqlastq: Receiver request queue (fifo) last quadlet. This bit is set if the quadlet available for reading from the **rreq** register is the last quadlet of the packet. This bit can be cleared by acknowledgment or by simply reading the quadlet.

sidqav: Self ID quadlet available. This bit is set if the quadlet available for reading from the **rreq** register is a quadlet of a self ID packet. This bit can be cleared by acknowledgment or by simply reading the quadlet available.

rreqqrderr: Receiver request queue (fifo) read (synchronization) error. This bit gets set when the CPU tries to read from an empty **rreq** register or after a bus reset.

rreqqfull: Receiver request queue (fifo) full. The request queue (fifo) did become full

rrspqqav: Receiver response queue (fifo) quadlet available. This bit indicates that a quadlet of a packet queued in the response fifo is available for reading from the **rrsp** register. This bit can be cleared by acknowledgment or by simply reading the quadlet.

rrspqlastq: Receiver response queue (fifo) last quadlet. This bit is set if the quadlet available for reading from the **rrsp** register is the last quadlet of the packet. This bit can be cleared by acknowledgment or by simply reading the quadlet.

rrspqrderr: Receiver response queue (fifo) read (synchronization) error. This bit gets set when the CPU tries to read from an empty **rrsp** register or after a bus reset.

rrspqfull: Receiver response queue (fifo) full. The response queue (fifo) did become full.

All events have an interrupt enable bit whose name equals the event prefixed by ’**e**’.

The memory status indication bits are (for diagnostic purposes):

rreqqf: Receiver request queue (fifo) full. The fifo is currently full.

rreqqaf: Receiver request queue (fifo) almost full. The fifo has precisely space for one more quadlet.

rreqq5av: Receiver request queue (fifo) five available. The fifo has at least space for five more quadlets.

rreqqe: Receiver request queue (fifo) empty. The fifo is empty.

rrspqf: Receiver response queue (fifo) full. The fifo is currently full.

rrspqaf: Receiver response queue (fifo) almost full. The fifo has precisely space for one more quadlet.

rrspq5av: Receiver response queue (fifo) five available. The fifo has at least space for five more quadlets.

rrspqe: Receiver response queue (fifo) empty. The fifo is empty.

7.3 Asynchronous transaction examples

This section gives two examples of performing a request transaction with the AVLink chip. Note that these are only examples and that many more possibilities for accomplishing the same transaction exist, e.g. all events can be detected by using the chip’s interrupt logic or by polling, error indications can be checked between sub actions or at the end of a task depending on the granularity of program code.

The first example uses synchronization on a packet basis. This is the most efficient method with a minimal amount of synchronization overhead. This method is the best choice if bus traffic is low and CPU accesses are slow (relatively to the chip). The second example uses synchronization on a quadlet basis, this method is safe and should always work, but introduces a lot more synchronization overhead.

For these examples, in a normal case after a power reset followed by a bus reset some initialization of the chip is necessary. The transmitter’s **maxrc** and **tos/tof** fields should be set to convenient values. These values depend on the application and the performance of the responder node. Also the **atxrst** bit must be cleared to enable the transmitter. For simplicity, the receiver’s **arxall** bit is expected to be cleared. In this case the receiver doesn’t store unsolicited responses in the fifo and the CPU isn’t bothered with them. Also the **rreqqrderr** and **rrspqrderr** status bits which were set after the bus reset need to be cleared to enable read accesses to the receiver fifos. In both examples there is no checking for eventual bus resets, these are expected to be handled by an exception handler. At each bus reset all asynchronous transactions must be cancelled and everything must be re-initialized.

7.3.1 First example: synchronizing on a whole packet:

After initialization the CPU can start writing the request packet of N quadlets to the request fifo by writing the first (N-1) quadlets to the **tx_rq_next** register and the last one to the **tx_rq_last** register of the chip. After the CPU has written the entire packet, it must check the **treqqwrerr** status bit. If this bit is set there has been a synchronization error, the CPU’s write accesses where too fast to be handled by the chip and the packet has been purged. After this event has been detected the CPU must acknowledge (clear) this bit by writing a value of one to it, and can try to rewrite the entire packet. If the **treqqwrerr** status bit didn’t get set the packet is now in the fifo and scheduled for transmission. The CPU must wait for a confirmation which is delivered to the receiver request queue at the end of the packet’s transmission(s) on the bus. This confirmation (one quadlet) will be indicated by the **rreqqqav** and **rreqqlastq** status bits both being set and can then be read from the **rreq** register. The confirmation quadlet contains information about the status of the transaction. In a normal case the status indicated by the request confirmation will be response pending. This means that the responder node will respond to the received request some time later.

Now two things can happen:

One, the responder node doesn’t respond within the time specified by the **tos/tof** value in which case a time out event will be indicated by setting the **timeout** status bit or

Two, the response is received in time, which will be indicated by the **rcvdrsp** status bit.

Each of these bits if set needs to be acknowledged by the CPU. At the same time as the **rcvdrsp** status bit is set the **rrspqqav** status bit will be set indicating that the received response packet can be read quadlet by quadlet from the **rrsp** register. In between reading quadlets from the **rrsp** register the CPU must check the **rrspqlastq** status bit to detect the last quadlet of the packet (unless the packet size is known in advance) and also the **rrspqrderr** for synchronization errors. Note that in case of a synchronization error there will never be a last quadlet status indication. In the case of a synchronization error the packet can be reread after acknowledging the **rrspqrderr** status bit.

7.3.2 Second example: synchronizing on each quadlet

After initialization the CPU can start writing the request packet of N quadlets to the request fifo by writing the first (N-1) quadlets to the **tx_rq_next** register and the last one to the **tx_rq_last** register of the chip. Between each two consecutive quadlet writes the CPU must wait for the **treqqwr** status bit to get set. After the **treqqwr** status bit for the last quadlet gets set, the CPU will have to acknowledge it explicitly (for the other quadlets it was automatically cleared by writing the next quadlet to the **tx_rq_next** register).

The packet is now in the fifo scheduled for transmission and the CPU must wait for a confirmation which is delivered to the receiver request queue at the end of the packet’s transmission(s) on the bus. This confirmation (one quadlet) is indicated by the **rreqqqav** and **rreqqlastq** status bits both being set and can then be read from the **rreq** register. The confirmation quadlet contains information about the status of the transaction. In a normal case the status indicated by the request confirmation will be response pending. This means that the responder node will respond to the received request some time later.

Now two things can happen:

One, the responder node doesn’t respond within the time specified by the **tos/tof** value in which case a time out event will be indicated by setting the **timeout** status bit or

Two, the response is received in time, which is indicated by the **rcvdrsp** status bit.

Each bit if set needs to be acknowledged by the CPU. At the same time as the **rcvdrsp** status bit is set the **rrspqqav** status bit will be set indicating that the first quadlet of the response packet can be read from the **rrsp** register. Reading the quadlet will clear the **rrspqqav** status bit and the CPU must check this bit together with the **rrspqlastq** in between each quadlet read to detect the next available quadlet in the **rrsp** register and whether it's the last quadlet of the packet.

7.4 Asynchronous packet formats

These packet formats are described in the PDI1394L11 data sheet. Always make sure you ahve the latest data sheet.

8 The register map

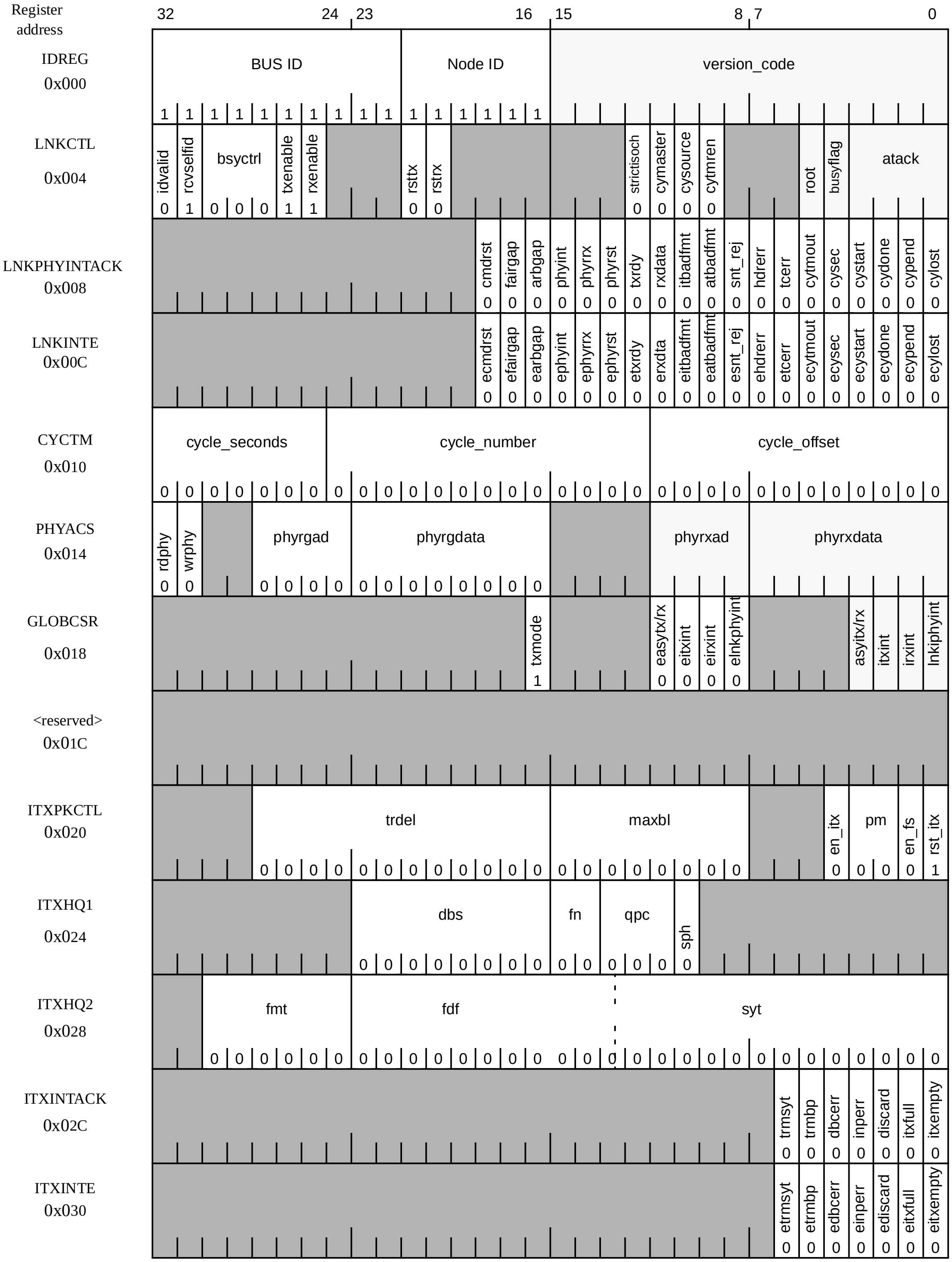
Registers are 32 bits (quadlet) wide and all accesses are always done on quadlet basis. This means that it is not possible to write just the lower 8 bits for example. Such as modification always requires a read and a write cycle. The values written to undefined fields/bits are ignored and thus don’t care.

A full bitmap of all registers is given on the next 3 pages. The meaning of shading and bit cell values is as follows:

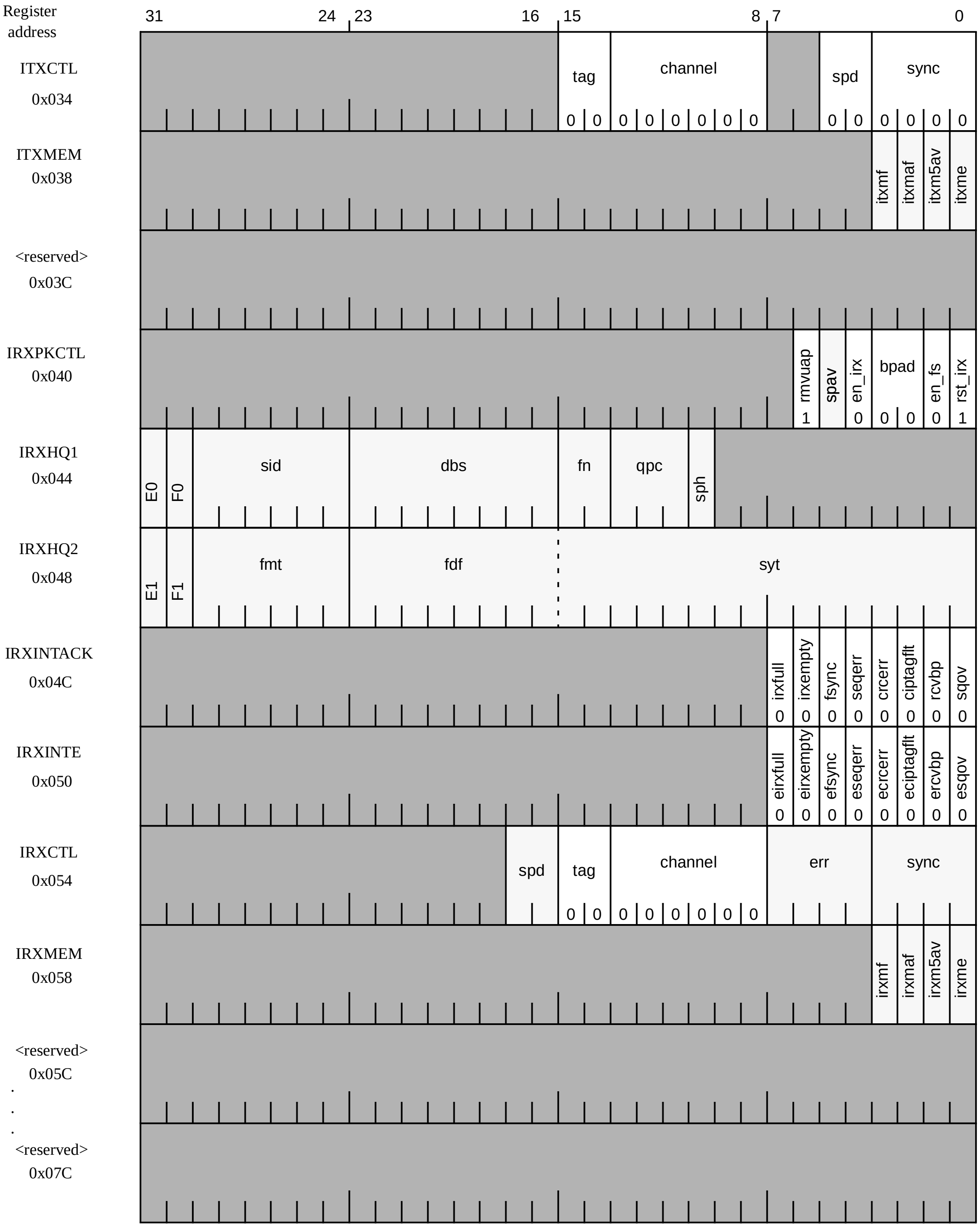
A bit/field with no name written in it and dark shading (▓) is reserved and not used.

A bit/field with a name in it and light shading (░) is a read only (status) bit/field.

A one bit value (0 or 1) written at the bottom of a writable (control) bit is the default value after power on reset.



Register address	Fields (bit 32 … 0)	Reset values
IDREG 0x000	BUS ID | Node ID | version_code	1 1 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 |
LNKCTL 0x004	idvalid | rcvselfid | bsyctrl | txenable | rxenable | rsttx | rstrx | strictisoch | cymaster | cysource | cytmren | root | busyflag | atack	0 | 1 | 0 0 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
LNKPHYINTACK 0x008	cmdrst | fairgap | arbgap | phyint | phyrrx | phyrst | txrdy | rxdata | itbadfmt | atbadfmt | snt_rej | hdrerr | tcerr | cytmout | cysec | cystart | cydone | cypend | cylost	0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
LNKINTE 0x00C	ecmdrst | efairgap | earbgap | ephyint | ephyrrx | ephyrst | etxrdy | erxdta | eitbadfmt | eatbadfmt | esnt_rej | ehdrerr | etcerr | ecytmout | ecysec | ecystart | ecydone | ecypend | ecylost	0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
CYCTM 0x010	cycle_seconds | cycle_number | cycle_offset	0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0
PHYACS 0x014	rdphy | wrphy | phyrgad | phyrgdata | phyrxad | phyrxdata	0 | 0 | 0 0 0 0 | 0 0 0 0 0 0 0 0 | |
GLOBCSR 0x018	txmode | easytx/rx | eitxint | eirxint | elnkphyint | asyitx/rx | itxint | irxint | lnkiphyint	1 | 0 | 0 | 0 | 0 | | | |
<reserved> 0x01C		
ITXPKCTL 0x020	trdel | maxbl | en_itx | pm | en_fs | rst_itx	0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 | 0 0 | 0 | 1
ITXHQ1 0x024	dbs | fn | qpc | sph	0 0 0 0 0 0 0 0 | 0 0 | 0 0 0 | 0
ITXHQ2 0x028	fmt | fdf | syt	0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0
ITXINTACK 0x02C	trmsyt | trmbp | dbcerr | inperr | discard | itxfull | itxempty	0 | 0 | 0 | 0 | 0 | 0 | 0
ITXINTE 0x030	etrmsyt | etrmbp | edbcerr | einperr | ediscard | eitxfull | eitxempty	0 | 0 | 0 | 0 | 0 | 0 | 0


Register address
31
24
23
16
15
8
7
0
ITXCTL 0x034
tag
0 0
channel
0 0 0 0 0 0
spd
0 0
sync
0 0 0 0
ITXMEM 0x038
itxmf
itxmaf
itxm5av
itxme
<reserved> 0x03C
IRXPKCTL 0x040
rmvuap
1
spav
en_irx
0
bpad
0 0
en_fs
0
rst_irx
1
IRXHQ1 0x044
E0
F0
sid
dbs
fn
qpc
sph
IRXHQ2 0x048
E1
F1
fmt
fdf
syt
IRXINTACK 0x04C
irxfull
0
irxempty
0
fsync
0
seqerr
0
crcerr
0
ciptagflt
0
rcvbp
0
sqov
0
IRXINTE 0x050
eirxfull
0
eirxempty
0
efsync
0
eseqerr
0
ecrcerr
0
eciptagflt
0
ercvbp
0
esqov
0
IRXCTL 0x054
spd
tag
0 0
channel
0 0 0 0 0 0
err
sync
IRXMEM 0x058
irxmf
irxmaf
irxm5av
irxme
<reserved> 0x05C
.
.
.
<reserved> 0x07C

Register address	31							24	23							16	15							8	7							0
ASYCTL 0x080										arxrst	atxrst	arxall	maxrc				tos			tof												
										0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0
ASYMEM 0x084															trspqidle	treqqidle	rrspqf	rrspqaf	rrspq5av	rrspqe	rreqqf	rreqqaf	rreqq5av	rreqqe	trspqf	trspqaf	trspq5av	trspqe	treqqf	treqqaf	treqq5av	treqqe
TX_RQ_NEXT 0x088	first/middle quadlet of packet for transmitter request queue (write only)																															
TX_RQ_LAST 0x08C	last quadlet of packet for transmitter request queue (write only)																															
TX_RP_NEXT 0x090	first/middle quadlet of packet for transmitter response queue (write only)																															
TX_RP_LAST 0x094	last quadlet of packet for transmitter response queue (write only)																															
RREQ 0x098	quadlet of packet from receiver request queue (transfer register)																															
RRSP 0x09C	quadlet of packet from receiver response queue (transfer register)																															
ASYINTE 0x0A0																rrspqfull 0	rreqqfull 0	sidqav 0	rrspqlastq 0	rreqqlastq 0	rrspqrderr 0	rreqqrderr 0	rrspqqav 0	rreqqqav 0	timeout 0	rcvdrsp 0	trspqfull 0	treqqfull 0	trspqwrerr 0	treqqwrerr 0	trspqwr 0	treqqwr 0
ASYINTE 0x0A4																errspqfull 0	erreqqfull 0	esidqav 0	errspqlastq 0	erreqqlastq 0	eirrspqrderr 0	erreqqrderr 0	errspqqav 0	erreqqqav 0	etimeout 0	ercvdrsp 0	etrspqfull 0	etreqqfull 0	etrspqwrerr 0	etreqqwrerr 0	etrspqwr 0	etreqqwr 0
<reserved> · 0x0A8 · ·																																
<reserved> 0x0F8																																

9 The host interface

The host interface allows an 8 bit CPU to access all registers and the asynchronous packet queues. It is specifically designed for an 8051 microcontroller but can also be used with other CPUs. The register map presented in chapter 8 shows that there are 64 register addresses (for quadlet wide registers). To access bytes rather than quadlets the address spaces is 256 bytes, requiring 8 address lines.

The use of an 8 bit interface introduces an inherent problem that must be solved: register fields can be more than 8 bits wide and be used (control) or changed (status) at every internal clock tick. If such a field is accessed through an 8 bit interface it requires more than one read or write cycle, and the value should not change in between to maintain consistency. To overcome this problem accesses to the chip's internal register space are always 32 bits, and the host interface must act as a converter between the internal 32 bit accesses and external 8 bit accesses. This is where the shadow registers come in.

9.1 Read accesses

To read an internal register the host interface can make a snapshot (copy) of that specific register which is then made available to the CPU 8 bits at a time. The register that holds the snapshot copy of the real register value inside the host interface is called the *read shadow register*. During a read cycle address lines A0 and A1 are used to select which of the 4 bytes currently stored in the *read shadow register* is output onto the CPU data bus. This selection is done by combinatorial logic only, enabling external hardware to toggle these lines through values 0 to 3 while keeping the chip in a read access mode to get all 4 bytes out very fast (in a single extended read cycle), for example into an external quadlet register.

This solution requires a control line to direct the host interface to make a snapshot of an internal register when needed, as well as the internal address of the target register. The register address is connected to input address lines A2..A7, and the update control line to input address line A8. To let the host interface take a new snapshot the target address must be presented on A2..A7 and A8 must be raised while executing a read access. The new value will be stored in the *read shadow register* and the selected byte (A0, A1) appears on the output.

Note 1: it is not required to read all 4 bytes of a register before reading another register. For example, if only byte 2 of register 21 is required a read of byte address 256 + (4 x 21) + 2 = 342 is sufficient.

Note 2: the update control line does not necessarily have to be connected to the CPU address line A8. This input could also be controlled by other means, for example a combinatorial circuit that activates the update control line whenever a read access is done for byte 0. This makes the internal updating automatic for quadlet reading.

Note 3: reading the bytes of the read shadow register can be done in any order and as often as needed.

9.2 Write accesses

To write to an internal register the host interface must collect the 4 byte values into a 32 bit value and then write the result to the target register in a single clock tick. This requires a register to hold the 32 bit value being compiled until it is ready to be written to the actual target register. This temporary register inside the host interface is called the *write shadow register*. During all write cycles address lines A0 and A1 are used to select which of the 4 bytes of the *write shadow register* is to be written with the value on the CPU data bus. Only one byte can be written in a single write access cycle.

This solution requires a control line to direct the host interface to copy the *write shadow register* to the actual destination register when ready, as well as the internal address of that register. The destination register address is connected to input address lines A2..A7, and the update control line to input address line A8. To let the host interface make the internal transfer the target address must be presented on A2..A7 and A8 must be raised while executing a write access. The current value on the CPU data bus will be stored in the *write shadow register* at the selected byte (A0, A1) and the result will be copied into the specified destination register.

Note 1: it is not required to write all 4 bytes of a register: those bytes that are either reserved (undefined) or don’t care do not have to be written in which case they will be assigned the value that was left in the corresponding byte of the *write shadow register* from a previous write access. For example, to set the interrupt enable mask for the isochronous transmitter (address 12) a single byte write to location 256 + (4 x 12) + 3 = 307 is sufficient (see below for byte order in quadlets).

Note 2: the update control line does not necessarily have to be connected to the CPU address line A8. This input could also be controlled by other means, for example a combinatorial circuit that activates the update control line whenever a write access is done for byte 3. This makes the internal updating automatic for quadlet reading.

Note 3: writing the bytes of the read shadow register can be done in any order and as often as needed (new writes simply overwrite the old value).

9.3 Byte order

The bytes in each quadlet are numbered 0..3 from left (most significant) to right (least significant) as shown in figure 22. To access a register at internal address N the CPU should use addresses E:

E = 4 N	; to access the upper 8 bits of the register.
E = 4 N + 1	; to access the upper middle 8 bits of the register.
E = 4 N + 2	; to access the lower middle 8 bits of the register.
E = 4 N + 3	; to access the lower 8 bits of the register.

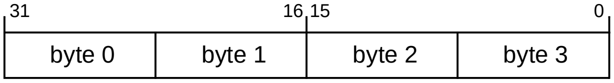


Figure 4: Byte order in quadlets as implemented in the host interface

9.4 Accessing the packet queues

Although entire incoming packets are stored in the receiver buffer memory they are not randomly accessible. These buffers act like fifos and only the frontmost (oldest) data quadlet entry is accessible for reading. Therefore only one location (register address) is allocated to each of the two receiver queues. Reading this location returns the head entry of the queue, and at the same time removes it from the queue, making the next stored data quadlet accessible.

With the current host interface such a read is in fact a move operation of the data quadlet from the queue to the *read shadow register.* Once the data is copied into the *read shadow register* it is no longer available in the queue itself so the CPU should always read all 4 bytes before attempting any other read access (be careful with interrupt handlers for AVLink!).

A similar argument applies to the transmitter queues. Data cannot be written arbitrarily, but only to the next available free location. Since the transmitter needs to know when the packet is complete (all data stored in memory, so that it may start the arbitration process on the 1394 bus) two separate register locations are reserved per transmitter queue: one to write all but the last packet quadlet to, and one to write the last quadlet of every packet to. Writing to any of these register locations stores the data in the queue and makes the next memory location accessible for writing.

Note: because of the way it is implemented memory access is not always immediate; consequently it may take some time before the next data quadlet in the queue is accessible after reading or writing the current one. Status flags are provided to the CPU to indicate availability.

9.5 The CPU bus interface signals

The CPU interface is directly compatible with an 8051 microcontroller. It uses a separate $\overline{\text{RD}}$ and $\overline{\text{WR}}$ inputs and a $\overline{\text{CS}}$ chip select line, all of which are active low. As explained before there are 9 address inputs (A0..A8) and of course 8 data in/out lines D0..D7. An active low $\overline{\text{INT}}$ output is used to signal interrupts to the CPU.

The CPU is not required to run at a clock that is synchronous to the 1394 base clock. The control signals will be resampled by the host interface before being used internally.

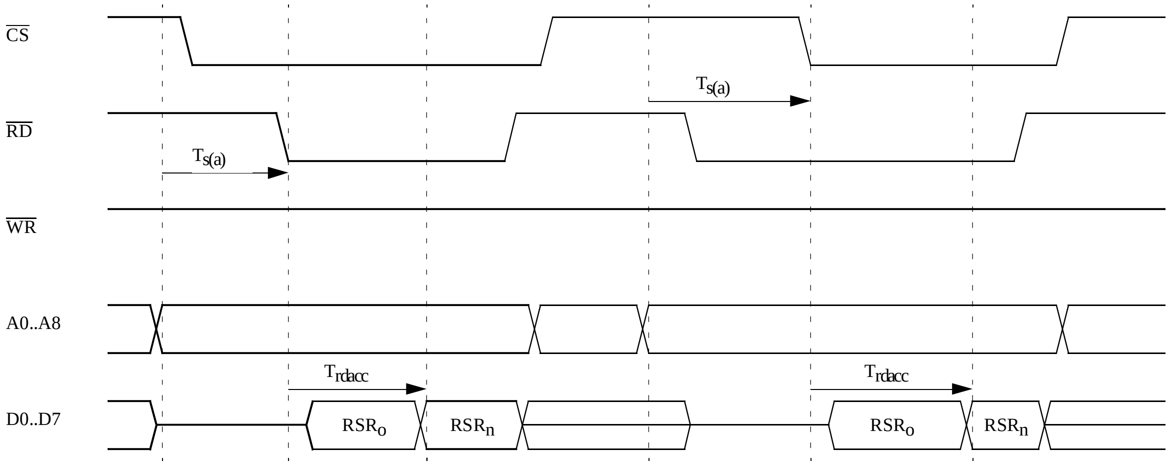


Figure 5: Read cycle signal timing (2 independent read cycles).

The read cycle timing shown in figure 23 is just an indication, for exact data refer to the data sheets. Basically, an access through the host interface start when $\overline{\text{CS}}$ = 0 and either $\overline{\text{WR}}$ = 0 or $\overline{\text{RD}}$ = 0. Typically the chip select signal is derived from the upper address lines of the CPU (address decode stage), but it could also be connected to e.g. a port pin of the CPU to avoid the need for an external address decoder in very simple CPU systems. When both $\overline{CS}$ = 0 and $\overline{RD}$ = 0 the host interface will start a read access cycle, so the cycle is triggered at the falling edge of either $\overline{CS}$ or $\overline{RD}$, whichever is later.

Very shortly after the start of the cycle (a few ns.) the selected byte in the *read shadow register* will be output (indicated as RSR_o). If A8 is asserted then the target register value will be copied into the *read shadow register,* leading to a new value RSR_n some time later in the read cycle. If A8 is negated then the *read shadow register* will not change and $RSR_n = RSR_o$.

The setup time of address lines A2..A8 to the start of the read cycle $T_{s(a)}$ is meant to be zero, but actual value depends on gate level implementation and is provided in the electrical data sheets. The access time during a read cycle T_{rdacc} is longest when a fresh register copy must be made from the target register and this time is designed to be no more than 105 ns, although the actual value may be different due to gate level implementation details. When data is read directly from the *read shadow register* without updating it the access time is much faster (possibly as low as 15 ns).

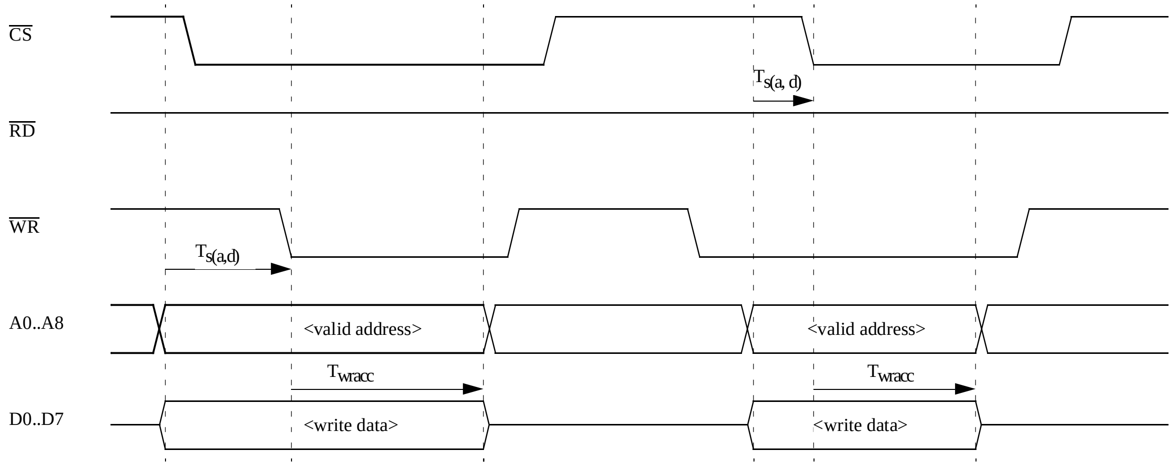


Figure 6: Write cycle signal timing (2 independent write cycles).

The timing of signals for write accesses, as shown in figure 24, is just an indication. Again the address, and this time also the data inputs, require a setup time before starting the access cycle $T_{s(a,d)}$ which is intended to be zero. The write access time T_{wracc} is designed to be 105 ns for any type of access (with or without register copy). The write access cycle starts when $\overline{CS}$ = 0 and $\overline{WR}$ = 0.

If a register copy from *write shadow register* to destination register is requested by asserting A8 then this will not increase the access time (the copy will be done "off-line" if needed).

Note: The time between the end of any access and the start of the next access must be at least 42 ns.

Note: When A8 = 0 for either write or read access the address bits A2..A7 are ignored.

Note: If both $\overline{WR}$ = 0 and $\overline{RD}$ = 0 while $\overline{CS}$ = 0 then a write cycle takes place.

10 Changes from first silicon

Although this document was intended to describe the features of a new (future) avlink chip it is possible that the first test silicon, with some internal bug fixes, will become the initial product release. This makes it more important to understand the differences between what is described in this document and what is in the first test silicon and what is in first product release (from a user perspective).

After first silicon tape-out some errors were discovered that could cause unexpected behavior of the avlink chip. Some of these can be worked around in software, and some cannot. These problems have been fixed without changing any of the external interfaces (including the programming model or the register bits) other than that work-arounds are no longer required.

A major change was introduced in the asynchronous packet interface where packet confirmations are no longer made available in a single register but queued in the receiver packet fifos instead. This requires a different software structure in the lowest level (driver) for avlink. It is possible to write software that can be adapted from first test silicon to latest design with only minor changes in the driver, but it requires careful structuring and knowledge of how confirmations are handled in both designs.

The following sections describe the relevant changes in more detail.

10.1 Isochronous transmitter changes

The following text describes the situation/errors in first test silicon.

10.1.1 SYT interrupt

The SYT interrupt bit (indicating that a valid SYT stamp has been transmitted on the bus) will be set immediately upon deactivation of the reset control bit. To use this interrupt in software it must be acknowledged after deactivation of reset. This is a minor inconvenience and is fixed in the avlink product release.

10.1.2 Data formatting with DBS=0

Setting the packing parameter DBS to 0, which is supposed to generate data blocks of size 256 quadlets does not work in first test silicon.

10.1.3 Data buffering during bus resets and lost cycles

When the node is cycle master and the transmitter somehow does not get the opportunity to transmit a packet (e.g. because of a bus reset) the data will accumulate. If the transmitter cannot catch up and these events keep occurring then eventually buffer overflows may occur and/or time stamps may expire before they are transmitted. This problem is fixed in the avlink product release.

10.2 Isochronous receiver changes

10.2.1 Data formatting with DBS=0

The isochronous receiver also has a problem processing IEC61883 data formatted with DBS = 0. Do not use it with first test silicon. The fix is implemented in the avlink product release.

10.2.2 Improper data handling after receiving corrupted isochronous packet

It is possible that isochronous packets are interrupted/destroyed midway by a bus reset. The first test silicon receiver will incorrectly handle such a situation, causing incorrect data on the AV interface and possible incorrect timing (for time stamped data). The problem is fixed in product release of avlink: the interrupted source packet will be discarded as if a sequence error had occurred.

10.3 Asynchronous packet interface changes

10.3.1 Blocking transmitter queue

After sending a request packet and receiving an ACK_PENDING in return the transmitter waits until it receives the proper response packet or a split transaction time-out occurs. Meanwhile if there is a packet waiting to be transmitted in the response queue it is ignored. This blocking could lead to "temporary deadlocks" (for example if 2 avlink nodes more or less simultaneously send a request to each other and then wait for the response), which will resolve after the time-out.

This error is fixed in the avlink product release.

10.3.2 Confirmation register

This is a problem only for nodes that need to handle multiple outstanding transactions. In order to handle outgoing transactions correctly software needs some kind of indication about what eventually happened to the transaction packet (received with crc error, type error, no error, etc.). This information is usually available from the ACK code returned by the receiving node and the indication generated by avlink to the application (CPU) is called a confirmation. Avlink generates two types of confirmations: request confirmations and response confirmations, each for the corresponding type of transaction. The problem with first test silicon is that it has only one register field for each of these two confirmations. An interrupt can indicate availability of a new confirmation but if the application does not retrieve the information quickly (within a few microseconds) it could be overwritten with the next confirmation (for the next packet).

The work around is never to write a packet to any of the transmitter queues until the confirmation for the previous packet written into the same queue has been read. This obviously eliminates the possibility for multiple outstanding request transactions. None of the queues can contain more than one packet in this case.

The fix for this problem is to store confirmations in the corresponding receiver queue and not in a register. The impact on software is considerable because (1) the register map changes as the old confirmation bits become defunct, and (2) confirmations must be retrieved from one of the queues where they may be "hidden" behind other packets already queued which must therefore be read first.

10.3.2.1 Transaction data confirmation formats

After a packet from one of the queues has been transmitted, the asynchronous transmitter assembles a so called confirmation (see figure 7) which is used to confirm the result of the transmission to the higher layers. Separate confirmations are assembled for request and response transmissions. Request confirmations are written into the request queue and response confirmations are written into the response queue.

<table>
<tr><td>31 30 29 28</td><td>27 26 25 24</td><td>23 22 21 20</td><td>19 18 17 16</td><td>15 14 13 12</td><td>11 10</td><td>9 8</td><td>7 6 5 4</td><td>3 2 1 0</td></tr>
<tr><td colspan="4">destinationID</td><td colspan="2">tLabel</td><td>00_2</td><td>1000_2</td><td>conf</td></tr>
</table>

Figure 7: Request and response confirmation format

The meaning of the conf code is as follows:

Code:	Meaning:
0:	Non broadcast packet transmitted, addressed node returned no acknowledge
1:	Broadcast packet transmitted or non broadcast packet transmitted for which the addressed node returned an acknowledge complete.
2:	Non broadcast packet transmitted, addressed node returned an acknowledge pending.
4:	Retry limit exceeded, destination node hasn’t accepted the non broadcast packet within the number of max. retry count.
D_{16}:	Acknowledge data error received (transaction complete)
E_{16}:	Acknowledge type error received (transaction complete)

All other values are reserved values.

For every packet written in a transmitter queue by the CPU there will be precisely one confirmation written in the corresponding receiver queue by the AV layer logic.

10.3.3 Reset control for asynchronous receiver logic and queues missing

The logic (and queues) of the asynchronous receiver in the AV layer are automatically reset at power-on reset and at every bus reset, but there is no way to do this under CPU control (other than to generate a bus reset). Such a CPU controlled reset is probably not required but for safety and compleness a control bit is added in the avlink product release. To prevent the asynchronous receiver from coming out of a reset state in the middle of an incoming packet the bit will auto clear at an appropriate time (i.e. when link layer receiver is idle).

10.3.4 Status & interrupt flags

The flags used to indicate "memory full" and the flags used to indicate "queue idle" are located in the interrupt status register (ASY_CSR2 bits 0, 1, 4, 5, 12 and 13 in first test silicon). When one of these bits changes state from 0 to 1 and the corresponding interrupt enable bit is set then the INT_N output will be activated (if not already active). The interrupt can then be acknowledged by the CPU by writing a ’1’ to the same bit. This will clear the interrupt on the INT_N output unless there are more active interrupts that still need to be acknowledged. However the indicated status bit itself will not be cleared since the value returned in the status bit is not the interrupt status bit itself but the signal that originally lead to the interrupt status being raised.

These flags now behave in a manner consistent with all other interrupt bits.

10.3.5 Memory status flags not available

Somewhat related to the previous problem, this one is just a minor deficiency at most. The application has no knowledge of the memory status for the asynchronous packet fifos. It is not really clear if this information is useful to applications. The proposed fix implements a special (new) status register containing 4 status bits per memory queue (total 16) plus 2 queue idle bits that are located in bits 0 and 1 of ASY_CSR2 in first test silicon. The 4 memory status bits are: ’full’, ’almost full’, ’5 or more available’, and ’empty’. Because of the impact on user interface and the fact that it’s not a required change this one is not likely to be available in first avlink product release.

10.3.6 Phy packets may cause fifo overflow

This is caused by the link core. An incoming PHY configuration or Link-On packet is written in the receiver fifo (request queue) without regard of the fifo status. This was not anticipated by the design of the queue controllers in the AV layer. The consequence is that if the request queue is full or almost full (= 1 quadlet space available) and a PHY packet is received then the fifo will overflow and the administration will be corrupted. There is no good work around, other than to disable receipt of PHY packet (RcvSelfID = 0) but this will also disable receipt of Self-ID data which is important.

A similar problem exists with Self-ID packets. In this case the link core writes everything it receives in the request queue until the queue is completely full (or no more Self-ID data is received) and then appends a final status quadlet. This situation also results in a fifo overflow and corrupted administration but it can be detected because Self-IDs are only received in a special state of the system: immediately following a bus reset. If the first quadlet read is not 0x000000E0 (the header for Self-IDs and PHY packets) then something is wrong.

In any case, if the fifo overflows then the queue controller will not work properly. Unfortunately there is no reset control bit for the asynchronous receiver. The only way to reset the receiver is by means of a bus reset or power-on reset. If the overflow was caused by too many Self-IDs then the problem cannot be worked around.

These two errors in the link core were fixed in the avlink product release.

10.3.7 Partial asynchronous packets

Due to the way the link core handles incoming packets it is possible that packets with a data field (such as lock requests/responses and block write requests and block read responses) are only partially stored in the receiver queue if that queue becomes full during the receipt of the packet. The result in this case will always look like this: complete header + first N quadlets of data field + status quadlet. Since the packet could not be stored completely the link layer has sent a BUSY ACK back to the sender and this is indicated in the final status quadlet (ack_sent field). The CPU should discard this incomplete packet. The original sender will probably retransmit the packet later.

To handle this properly in first test silicon, software can read the entire header as usual (blind, or standard polling) but then it must proceed by reading every data quadlet separately and checking if the current quadlet is the last one stored for this packet (additional status flag polling).

The problem is fixed in the first avlink product release in such a way that partial packets are automatically removed (never confirmed). This has 3 beneficial effects: (1) Software for reading packets can be simplified, (2) CPU is not burdened by reading data that’s of no use anyway, and (3) fifo will never contain unnecessary data which increases the probability that the packet will fit when retried in the next fairness interval.

10.4 Other changes

10.4.1 Bus ID value after bus reset

The BUSID field in the ID register is not affected by bus reset in first test silicon. The latest design sets the BUS ID to 0x3FF automatically. For first silicon this must be done in software as soon as possible after bus reset but definitely before handling any kind of asynchronous packet in any direction (in/out). This is fixed in first avlink product release.

10.4.2 Link layer reset and enable bits

In first test silicon the reset bits for the link layer (RstTx and RstRx) are activated at power-on reset, and the enable bits (TxEnable and RxEnable) are deactivated at power-on reset. This puts the link layer in a continuous reset and disabled state until the CPU decides to put an end to this situation. A particular effect will probably be that the Self-IDs following the bus reset caused by this node’s power-up will be missed completely because the CPU generally takes a long(er) time to start up. Consequently, after the CPU starts up it must initialize avlink and then generate another bus reset so that it can receive the Self-ID data. This is not considered "nice behavior". The fix for this problem (in the avlink product release) is to automatically clear (deactivate) the reset bits after a few clock ticks and to default the enable bits to an active state immediately upon power-on reset.

10.4.3 EnableAT bit

The link core specifies that bit 19 of the first header quadlet of the first packet written into a queue after each bus reset must be ’1’. This is used to flush stale packets. Avlink has no use for this feature since the fifos are automatically flushed upon bus reset and the rest is up to software. The new design handles the ’bit 19 protocol’ with the link core automatically. The first test silicon however still requires that bit 19 of the first header quadlet is set to ’1’. Because of the way avlink works this bit can be set to ’1’ for all asynchronous packets (not just the first). For avlink product release this bit will be a don’t care.

10.4.4 Unsolicited response indication

The extra status bit ’U’ at bit location 4 in the status quadlet appended to every received asynchronous packet in the receiver queues is not available in first test silicon (there this bit is always ’0’). This is a new AV layer feature available in the avlink product release.

10.4.5 Cycle Timer Register does not increment when it is read by CPU

The CYCTM reloads its current value when the CPU performs a read cycle (with copy into shadow register) on it. This error is fixed in avlink product release.

10.4.6 The transmitter ready (TxRDY) interrupt fix

The TxRDY bit of the LNKPHYINTACK register (address 0x004) was never triggering an interrupt in the first silicon. This has been fixed in the avlink product release.

10.5 Register map changes

The register map and meaning of control and status bits for the latest design have been specified earlier in this document. First test silicon has a different register map, and the names of various registers and bits/fields are also different. The names used in the previous chapters in this document are, to the authors opinion, more consistent (both mutually and w.r.t. the functionality they represent) and easier to remember. Below is a more detailed description of the differences (other than just renaming).

10.5.1 GLOBCSR (address 0x018)

Old name: GENINT ; external byte address 0x018

- 4 new control bits have been added since first test silicon (bits 8..11) to function as master interrupt enables for an entire module of the avlink (easytx/rx, eitxint, eirxint, elnkphyint). Only the asynchronous interface has a master interrupt enable control bit in the first test silicon, and there it is located in ASY_CR1 bit 22 (INTENA). This control bit has been relocated to GLOBCSR bit 11 in the current design.

10.5.2 ITXINTACK, ITXINTE (internal addresses 0x02C and 0x030)

Old name: ITXINTCTL ; byte address 0x02C

- The 7 interrupt status/ack bits and the 7 interrupt enable (mask) bits are located in the same single register in first test silicon (status at bits 0..6, mask at bits 16..22). In the new design the status/ack bits are still in the same location, but the enable (mask) bits have been moved to a new register located at internal quadlet address 0x0C (one location above the status/ack bits) and are bit aligned to the status/ack bits (bit position 0). This is more consistent with the way interrupt status and mask are handled for the link layer module, allows these interrupts to be enabled/disabled or acknowledged using only a single byte write and is therefore expected to make things easier for software.

10.5.3 IRXINTACK, IRXINTE (internal addresses 0x04C, 0x050)

Old name: IRXCTL ; byte address 0x040

- The 8 interrupt enable bits for the IRX module were located in the same register that also contains several control bits for the receiver. To make this module more consistent with the link layer module and the ITX module these enable bits have been moved to a new register (IRXMSK) located one position above the IRXINT interrupt status/ack register in address space. The enable bits have also been shifted to bit position 0 to align them with the corresponding status/ack bits. This allows interrupts to be enabled/disabled or acknowledged with single byte writes and also the remaining controls bits in IRXCTL can be changed using a single byte write operation.

10.5.4 IRXCTL (internal address 0x054)

Old name: IRXISOCTL ; byte address 0x050

- Because of the insertion of a new interrupt mask register the remaining registers in the IRX module have been shifted one position up in the address space: IRXISO is identical to the old IRXISOCTL but its address changed from 0x050 to 0x054.

10.5.5 IRXMEM (internal address 0x058)

Old name: RXMEMS ; internal quadlet address 0x15, external byte address 0x054

- Because of the insertion of a new interrupt mask register the remaining registers in the IRX module have been shifted one position up in the address space: IRXMEM is identical to the old RXMEMS but its address changed from 0x054 to 0x058.

10.5.6 ASYCTL (internal address 0x080)

Old name: ASY_CR1 ; byte address 0x080

- The master interrupt control bit (INTENA) for the asynchronous interface (at bit position 22) has been removed in the new design. Its function is now implemented using a new control bit located in GLOBCSR.
- The bit previously occupied by INTENA is now used for ARXRST, an auto clearing reset control bit by means of which the CPU can reset the logic and queues of the asynchronous receiver in the AV layer. This bit should never be cleared by the CPU and always reads back as 0.

10.5.7 ASYMEM (internal address 0x084)

Old name: - - -

- A register with this function does not exist in first test silicon, but the address (0x084) is in use for another register (ASY_CR2) in first test silicon. In the new design it contains the 4 memory status bits (full, almost full, 5 available, empty) for each of the 4 asynchronous queues, plus the 2 idle indications for the transmit queue transfer registers. Some of these bits (the idle indications and the memory full status bits) are available in the first test silicon in bits 0, 1, 4, 5, 12 and 13 of ASY_CR2 at internal location 0x21, but since ASY_CR2 is primarily used for interrupt sources and interrupt enables they should not be placed there.

10.5.8 ASYINTACK, ASYINTE (internal addresses 0x0A0, 0x0A4)

Old name: ASY_CR2 ; byte address 0x084

- The interrupt status/ack bits and interrupt enable (mask) bits for the asynchronous interface are located in a single register in first test silicon (ASY_CR2) with a displacement of 17. Some of the bits have become defunct due to the reorganization of packet confirmation handling. For these reasons the old ASY_CR2 register has been split up in two new registers: one containing the interrupt status and acknowledge bits (ASYINTACK) and one containing the enable bits (ASYINTE). The enable bits are aligned to the corresponding interrupt status bits (both start at bit position 0). The defunct bits have been removed and new ones introduced as required by the new confirmation mechanism. Also the interrupt acknowledge now works the same for all modules: acknowledging an interrupt really clears the corresponding status bit until the next time such an event occurs.

10.5.9 - - - (internal address 0x0A0, 0x0A4)

Old names: RCV_TRQCF and RCV_TRPCF

- These registers contain the confirmation value for the request and response packets in first test silicon, but are no longer implemented in the new design. The locations (0x0A0 and 0x0A4) have been recycled for the new ASYINTACK and ASYINTE registers.

11 References

[1] IEEE Standard for a High Performance Serial Bus
IEEE Std 1394-1995; August 30, 1996;
IEEE Standards Department, 345 East 47th Street, New York, NY10017, USA.

[2] IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses
IEEE Std 1212-1991; July 28, 1992;
Published by IEEE Inc. 345 East 47th Street, New York, NY10017, USA.

[3] IEC 61883 Working draft: Digital Interface for Consumer Electronic Audio/Video Equipment.

[4] R.H.J. Bloks, The IEEE-1394 High Speed Serial Bus, Philips Journal of Research: special issue on digital television, Vol. 50, No. 1/2 1996 (ISSN 0165-5817)

[5] R.H.J. Bloks, IEEE-1394 Bus Management for IEC-61883 compliant devices, Philips Research Lab. Eindhoven/IST internal report, available from author (Philips internal use only).

Electronic references on 1394 in general:

[6] http://www.firewire.org/

[7] http://www.skipstone.com/

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS
Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

print code Date of release: 05-96

Document order number: 9397 750 02997