CONTENTS

INTRODUCTION

The Generic Addressable Bus Link aka GABus Link aka GAB-Link is a layered protocol stack designed for handling both unsecure and secure/reliable communication between a PC and a generic address/data bus.

The GAB-Link has been developed as a natural evolution of the TosNet Generic Interface Protocol (TGIP) and uTosNet Generic Interface Protocol (uTGIP) stacks.

SOME TGIP HISTORY

The TGIP stack where initially developed as the protocol of the existing PC<>TosNet interface the generic master node had several deficiencies/shortcomings. These shortcomings where lack of support for TosNet features such as reading slave registers, network registers as well as sending and receiving asynchronous data.
The generic master node protocol could be interfaced to a PC using either a simple serial (RS232) interface at low baud rate, with no parity or crc checking on the packets transferred, it could also be interfaced using a UDP/TCP socket, providing higher throughput but not additional TosNet features. The socket interface was handled using a Digi Connect ME 9210 ethernet enabled processor, this was interfaced to the FPGA running the TosNet node using a 4Mbit SPI interface, hence the maximal throughput of the socket connections where limited to this communication rate.

The TGIP were designed to be able to provide a full TosNet interface with access to all TosNet features as well as adding extra in form of a subscription service, making the TosNet master node (with TGIP) able to autonomously transmit subscribed register dwords to the connected PC at a prespecified rate.

An uTosNet edition of the TGIP stack was also created, called uTGIP. The uTGIP stack is based completely on the TGIP stack and the only difference between the two is that uTGIP uses a feature reduced version of the TGIP application layer. The uTGIP application layer is optimized for the smaller address space of the uTosNet shared memory.

Both the TGIP and uTGIP stacks were designed with modularity in mind making it possible to use various datalink and physical layers in order to achieve the desired transmission security/reliability and physical medium (UART, I2C, SPI, JTAG, FIFO, Ethernet etc.)

Initially a SPI version of the physical layer was made as it was deemed necessary to use a socket interface through the Digi Connect ME 9210 processor in order to get high data throughput. It was however discovered that the UART to USB bridge chips used on the FPGA boards was capable of baud rates between 1 to 3M baud and it's possible of getting some capable of up to 12M baud.

The FTDI FT232H High speed USB to UART/FIFO IC is actually capable of up to 12M baud UART interface and up to 40MByte/s using an 8-bit FIFO interface both modes can be accessed through a virtual COM (serial) port on the PC, making interfacing very easy and independent of the physical implementation of the interface on the FPGA.

As the SPI interface on the Digi processor has a max data rate of 4Mbit it is not desirable when compared to the high speed USB UART/FIFO bridges to use the SPI Digi interface. As equal and higher data rates easily can be achieved with a simpler solution, which also is easier to interface to from a PC.

TGIP/uTGIP PROBLEMS

From using the TGIP and uTGIP stacks a few shortcomings were identified:

The first and most obvious was the message/packet structure dictated by the datalink and application layers, which required any commands as well as data and error messages to be formatted using only Hexadecimal numbers (characters: 0-9, a-f and A-F). This made it difficult for a human to send and receive (understand) messages by hand from e.g. a simple terminal.

The second was the fact that the subscription feature was only implemented with one subscription group that only could take up to 32 addresses all of which thus only could uploaded at the same rate.

The third and most serious problem was the fact that the TGIP protocol was integrated directly into the TosNet core. This of course had the benefits of making it easily possible to implement features such as asynchronous data transfer as well as dedicated protocol commands for committing input and/or output registers in the TosNet double buffered memory scheme. But the problem created by this tight integration was inflexibility toward interfacing to both general IO and computation/control units on the same FPGA fabric as the TosNet core while still being able to support interfacing to a TosNet network.

A typical example of a TGIP (TosNet) setup can be seen in Figure 1. The TosNet core connect directly to the TosLink transceivers, hence the only way of connecting any local IO or computation/control units to the TosNet on the same FPGA fabric as the TGIP stack, is to instantiate an additional TosNet core in the FPGA as illustrated in Figure 2. This approach is highly inefficient in terms of logic usage and also undesirable as it increases the number of TosNet nodes in the system, lowering the network cycle frequency as well as decreasing the number of real individual physical network nodes.

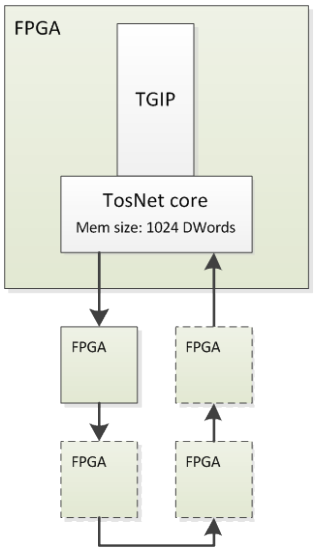

Figure 1: TGIP (TosNet) standard example

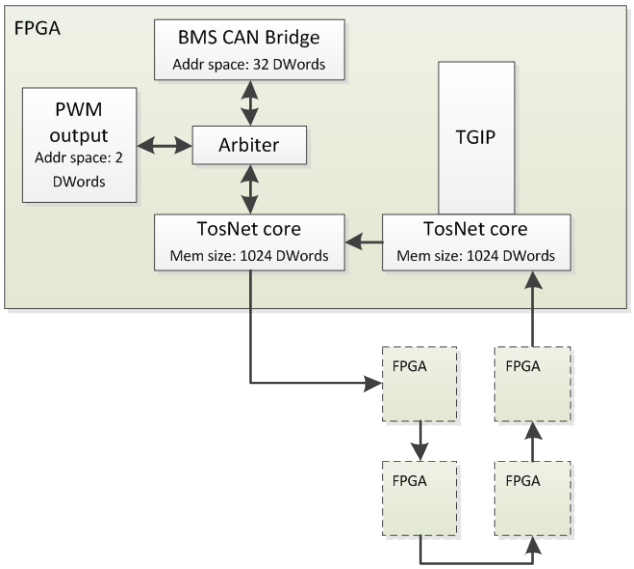

Figure 2: TGIP (TosNet) local IO example

Local IO and computation/control unit interfacing could isolated be solved by using the uTGIP stack. But this solution can't connect to a TosNet network very easily, it would become very involved to implement the commit in/out scheme and the solution would also scale poorly as the uTGIP stack can't be connected to several smaller sized memories but is integrated directly into one memory. This

quickly leads to wasted memory resources as well as the need for slow and arbitrated accesses to the single memory interface available to the logic, when more than one IO/control unit is present.

Figure 3 illustrates an uTGIP (uTosNet) setup with two individual cores: a PWM output core and a Battery Management System CAN Bridge core. The BMS core requires 32 DWords of memory space and the PWM core requires 2 DWords. As the uTGIP stack implements its accessible memory space as one contiguous memory unit, an arbiter is necessary for the PWM and BMS cores to both be able to access the memory. As the size of the uTGIP memory must be a power of 2 it needs to be 64 DWords large in order for the required 34 DWords to fit into it. Hence 30 DWords of memory space are wasted.

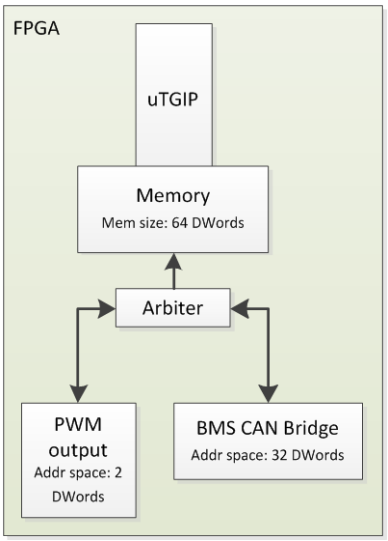

Figure 3: uTGIP (uTosNet) standard example

GAB-LINK FEATURES

The GAB-Link is like both the TGIP and uTGIP stacks a layered protocol stack designed for handling both unsecure and secure/reliable communication. But instead of interfacing directly to either a TosNet core or a memory like uTosNet it is designed for handling communication between a PC and a generic address/data bus. This approach solves the inflexibility and resource waste problems of both the TGIP and uTGIP stack implementations, at the expense of a slightly more general interface without the possibility of dedicated protocol commands for special purpose actions like for example commit in/out registers of double buffered memories, and transmitting/receiving asynchronous data. Such actions must instead be implemented at a higher abstraction level using general bus transactions.

The GAB-Link implements a subscription manager which supports up to 16 individually configurable subscription groups. Each subscription group is capable of holding up to 32 addresses and can be individually configured for requesting data publishing at integer multiple of the subscription manager base publish frequency. This allows up 512 addresses to be subscribed for real-time publishing. The GAB application layer protocol can in theory support up to 256 individual subscription groups each holding up to 255 addresses without requiring any redesign, but for any practical purposes these limits are out of practical range both due to needed logic consumption and the potentially enormous size and bandwidth usage of a publish message, and if all subscription groups request to publish at the same time.

The datalink layer and application layer has been redesigned to support a new more intuitive and human readable message/packet format. This new format allows arbitrarily long commands and data fields in a message, and the entire english alphabet in both lower and upper case ASCII characters (a-z and A-Z) can be used for the command part of a message, while the data part is restricted to the hexadecimal ASCII characters (a-f, A-F and 0-9).

The GAB-Link is like both the TGIP and uTGIP stacks designed with modularity in mind making it possible to use various datalink and physical layers in order to achieve the desired transmission security/reliability and physical medium (UART, I2C, SPI, JTAG, FIFO, Ethernet etc.)

WISHBONE COMPATIBLE MEMORY INTERFACE

The GAB-Link is as the name Generic Addressable Bus Link suggests an interface for generic bus access. But in order to actually make practical use of the GAB-Link it must interfaced to a BUS. Several standard BUS schemes for System on Chip (SoC) exist; the most prevalent is listed below:

- AMBA
  - Advanced eXtensible Interface (AXI)
  - Advanced High-performance Bus (AHB)
  - Advanced System Bus (ASB)
  - Advanced Peripheral Bus (APB)
  - Advanced Trace Bus (ATB).
- IBM Core connect
  - Processor Local Bus (PLB)
  - On-chip Peripheral Bus (OPB)
  - Device Control Register (DCR)
- Altera
  - Avalon Bus
- OCP International Partnership (OCP-IP)
  - Open Core Protocol (OCP)
- Open Cores Project
  - **Wishbone**

Even though the standard bus for the Xilinx EDK tool suite is the AMBA AXI bus, the Wishbone bus has been selected as the default bus technology for the initial work with the GAB-Link.
Without going into a lengthy discussion and evaluation of the different bus technologies, their advantages and disadvantages it should be noted that the main reason for choosing the Wishbone bus is that it is an royalty free open source SoC bus maintained under the Open Cores project, meaning that there exist a multitude of free high quality open source IP-Cores for this Bus. Another main reason is that while having relatively high performance the wishbone bus is simple to implement, making it relative easy to interface any wishbone compatible cores to for example PLB or AXI, the two most prevalent SoC BUS technologies with regard to Xilinx FPGA devices.

Figure 4 and Figure 5 shows how the GAB-Link (using a Wishbone bus) directly can emulate the TGIP and uTGIP stacks. From Figure 5 it can also be seen that the GAB-Link architecture doesn't have the resource waste associated with the uTGIP stack. This is the case as the GAB-Link architecture doesn't need to allocate any memory resources other than those explicitly needed.

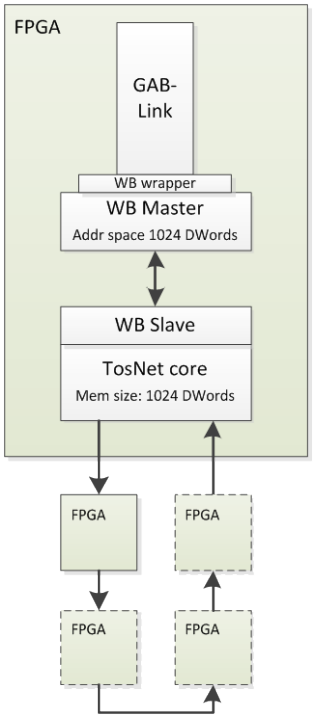

Figure 4: Wishbone GAB Link (TosNet)

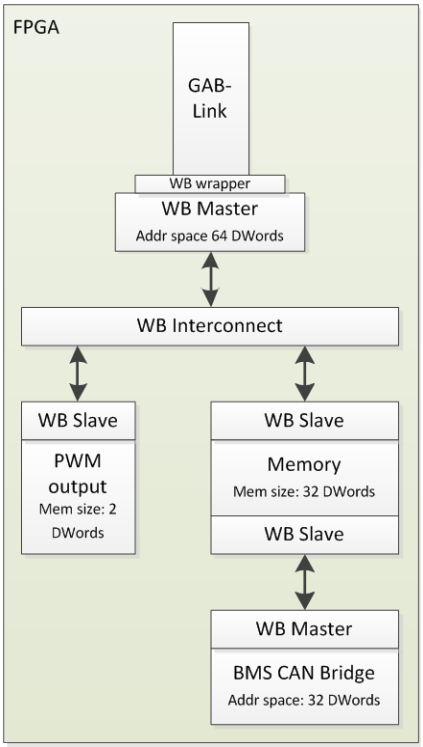

Figure 5: Wishbone GAB Link (''uTosNet'')

Figure 6 shows how the GAB-Link architecture easily and efficiently can combine the infrastructure of both the uTGIP and TGIP stacks without suffering from the deficiencies previously mentioned.

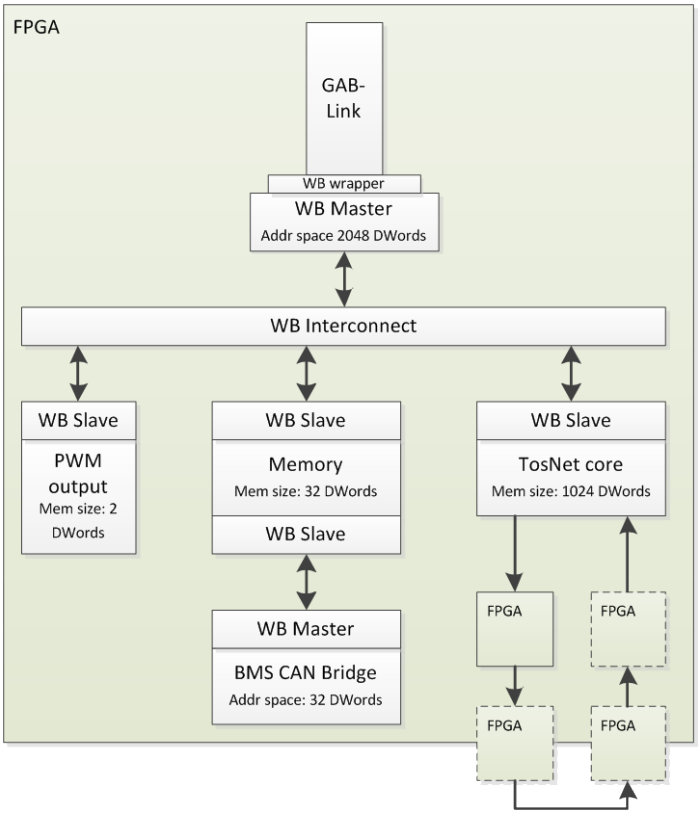

Figure 6: Wishbone GAB Link (TosNet+"uTosNet")

PROTOCOL STACK OVERVIEW

The GAB-Link stack consists of two mandatory and one optional layer. The mandatory layers are the physical layer and the application layer. The optional layer is the datalink layer.
Figure 7 shows an illustration of the GAB-Link stack. The main dataflow between the physical, datalink and application layers is done through FIFO's ensuring easy insertion/removal of additional layers. Depending on the physical layers implementation

PHYSICAL LAYER

The physical layer implement the logic necessary to interface the Datalink or Application layer to the interface medium.
The interface medium could be any type implementable in the digital logic of an FPGA such as: UART, I2C, SPI JTAG, FIFO or Ethernet interfaces, some requiring external signal conditioning logic.

DATALINK LAYER

The datalink layer and/or parts of it are optional; its presence/configuration depends on the physical layer implementation and whether or not data decoding/encoding and message checksum tests/generation is necessary.

The main task of the datalink layer is to ensure that only valid messages are passed on to the application layer. This makes it possible to implement the application layer without the need for it to rigidly verify the validity of an entire packet before beginning to process it, hence both enabling increased message processing speed and simplified error handling logic.

APPLICATION LAYER

The application layer implement the uTosNet Generic Interface Protocol execution unit, it interfaces directly to the uTosNet core and handles all requests that has passed through the physical and datalink layers successfully.

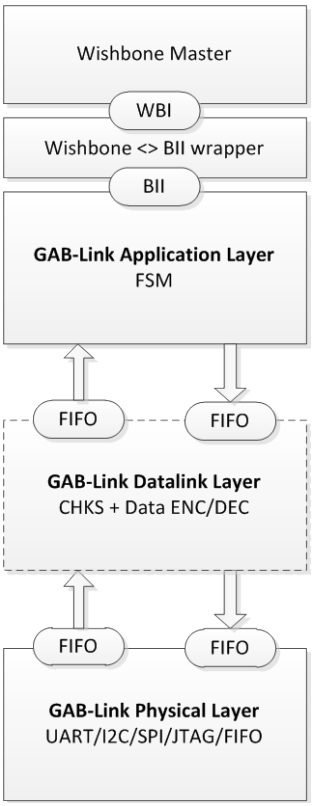

Figure 7: GAB-Link stack

GENERIC ADDRESSABLE BUS LINK PROTOCOL SPECIFICATION

The GAB-Link protocol is primarily designed to be used with a UART/byte oriented protocol and must be easily useable through a standard terminal program, hence the data encoding has been selected to be ASCII with all data/addresses etc. written in hexadecimal.

Request messages are sent in packets framed by the '#', '$' and '\n' (line feed) command code characters. Table 1 shows the valid packet command codes and allowed data separators.

Square braces: […] always encloses a single byte.
Regular braces: (…) are used to group bytes into logical structures.
Triple braces: […]]]] or (…))) indicate a variable number of bytes or groups of bytes.
Curly braces: *{...}* means the response or argument enclosed is optional

The packet format for a non-checksum request/response message is: [#](payload)[\n]

The packet format for a checksum request/response message is: [$](payload)[*]([chk1][chk2])[\n]

The packet format for a non-checksum publish message is: [%](payload)[\n]

The packet format for a checksum publish message is: [&](payload)[*]([chk1][chk2])[\n]

Packet command codes and data separators			
ASCII code	Hex Value	Packet CMD Type	Description
#	0x23	Request/Response	Marks the start of a Req/Rsp (no CHKS)
$	0x24	Request/Response	Marks the start of a Req/Rsp (with CHKS)
%	0x25	Publish message	Marks the start of a publish message (no CHKS)
&	0x26	Publish message	Marks the start of a publish message (with CHKS)
?	0x3F	Error message	Marks the start of an error message (no CHKS)
:	0x3A	cmd/data separator	Separates the payload command and data
*****	0x2A	CHKS marker	Marks the start of the checksum data.
\n	0x0A	All	Marks the end of any type of message
CR	0x0D	All	Marks the end of any type of message
TAB	0x09	All	Tab: allowed data separator
	0x20	All	Whitespace: allowed data separator
,	0x2C	All	Comma: allowed data separator
.	0x2E	All	Period: allowed data separator

Table 1: Packet Command Codes and Data Separators

The payload of a packet is always structured as a set of command bytes followed by multiple trailing data byte pairs (ASCII format):

payload = [cmd]]]{[:]([data_1][data_2]))}

Valid payload command characters are: [a..z] and [A..Z]

Valid payload data characters are: [0..9], [a..f] and [A..F]

When the GAB-Link stack receives a message the packet command codes and payload cmd bytes are not decoded by the datalink layer, but only marked with a special command bit. Data separators are simply thrown away and data bytes are decoded in pairs into a binary value, hence the limitation of only using the characters [0..9], [a..f] and [A..F] for data bytes and the requirement for them to appear in pairs.
Two ASCII bytes must be transmitted for each binary payload data byte:
To transmit the decimal value of <255>, the ASCII characters of <FF> must be transmitted this correspond to transmitting the hexadecimal value of <0x4646>

In case of transmission errors (parity/framing error in the UART version) from the physical layer the entire package must be disbanded (all rx fifo's will be flushed) and an errorcode must be sent as response, this must be enforced in both '$' and '#' messages.
In case of CHKS (checksum) error in a '$' message, the message must be disbanded (all rx fifo's must be flushed) and an error code must be sent as response.

The checksum is calculated from either a byte wise CRC8 or simple XOR operation on all payload cmd and data bytes after they have been decoded into binary format. The resulting checksum byte is then converted into ASCII (hex) code and transmitted between the '*' and '\n' characters. To ensure compatibility with windows terminals that normally doesn't send the '\n' (line feed) character the 'CR' (carriage return) character must also be interpreted as '\n'.

White-spaces, tabulator, semicolon and commas are allowed in messages as data delimiters/separators (will be ignored). Error codes from the datalink layer must always be sent as [?][errorcode]]][\n] messages (no checksum).

The application layer FSM may implement error handling that checks the validity of the command field and if applicable also the data field. Any error codes/messages must always be sent as [?][errorcode]]][\n]. An application layer error message is allowed to be inserted into an unfinished response if necessary ex.: [#]([R][:]([05][a1][ef][30])[?][errorcode]]][\n]

The application layer must frame a payload message with the appropriate datalink layer command codes: '#', '$', and '\n'. It must also separate the payload command bytes from the data bytes using a ':' (colon) character.

The Datalink layer must remove the '*' command code and the CRC checksum byte from the decoded message so the application layer only sees the packet framing codes '#'/'$', ':' and '\n' as well as the payload cmd and data bytes.
Likewise the Datalink layer must automatically calculate the CRC checksum and insert the * command code and the checksum byte into any response/publish message from the application layer framed with the '$' command code.

The application layer operates in pure binary format, hence in order for the application and datalink layers to be able to distinguish cmd and data payload bytes and identify the start and end of a message the packet framing codes passed between the datalink and application layers must be marked with a special control bit. The control bit is an additional bit in the data FIFO's between the DL and AL, thus the FIFO's must be 9 bit's wide.

GAB-LINK MESSAGES: REQUEST/RESPONSE/PUBLISH SYNTAX

All payload data bytes must be transmitted and received in hexadecimal format.

AL = Application Layer
DL = Datalink Layer (The datalink layer is not mandatory)

Table 2 shows all valid request command codes that can be transmitted to the GAB-Link application layer.

Hex Value[1]	ASCII value[2]	Payload CMD Type	Description
0x52	R	Command (AL)	Read
0x524D	RM	Command (AL)	Read Multiple
0x525349	RSI	Command (AL)	Read Subscription Info
0x524C43	RLC	Command (AL)	Read Link Configuration
0x57	W	Command (AL)	Write
0x575352	WSR	Command (AL)	Write Subscription Rate
0x575341	WSA	Command (AL)	Write Subscription Addresses
0x455053	EPS	Command (AL)	Enable Publish Service (without checksum)
0x455043	EPC	Command (AL)	Enable Publish service with Checksum
0x445053	DPS	Command (AL)	Disable Publish Service

Table 2: Payload Command Codes

Square braces: […] always encloses a single byte.
Regular braces: (…) are used to group bytes into logical structures.
Triple braces: […]]] or (…))) indicate a variable number of bytes or groups of bytes.
Curly braces: *{…}* means the response or argument enclosed is optional

Address and Data bytes	
([addr_MSB]]…LSB]) Address byte count	The number of bytes used for the address ([vaddr_MSB]]…LSB]) depend on the size of the GAB-Link address space. The address space can be between 1 and 32 bits, hence there can be between 1 and 4 address bytes in the address. Address bytes must always be transmitted MSB first!
([data_MSB]]…LSB]) Data byte count	The number of data bytes expected/transmitted for a single address: ([data_MSB]]…LSB]) depend on the size of the GAB-Link data space. The data space can be between 8 and 32 bits, hence there can be between 1 and 4 data bytes in a single variable. Data bytes must always be transmitted MSB first!

[1] Hex values shown only for upper case (capital) letters, but any combination using lower and upper case letters is valid.
[2] ASCII values shown only with upper case (capital) letters, but any combination using lower and upper case letters is valid.

READ LINK CONFIGURATION		
Request Syntax	*[RLC]*	
Request example (no chks)	Read Link Configuration: ASCII: #RLC\n HEX: 23 *524C43 0A*	
Request example (chks)	Read Link Configuration, chks=crc8: ASCII: $RLC*9B\n HEX: 24 524C43 2A 3942 *0A*	
Argument description	*RLC*	Command code = 0x524C43
Response syntax (success)	*[CNF][:][addr_bytes][,][data_bytes][,][rm_size][,][pub_mode][,][pub_base_freq][,][sg_cnt][,][sg_size][,][rate_bytes]*	
Argument description	*CNF*	Read success code = 0x434E46
	addr_bytes	Number of address bytes
	data_bytes	Number of data bytes
	rm_size	Maximum number of addresses in a RM (Read Multiple) request.
	pub_mode	Publish Mode: 0: Publish mode disabled 1: Prioritize incoming requests 2: Prioritize publish requests
	pub_base_freq	Publish base frequency
	sg_cnt	Number of subscription groups
	sg_size	Number of individual address subscriptions each subscription group can contain.
	rate_bytes	Number of bytes in the rate register in each subscription group.
Response syntax (error)	E_ICC	Error code (AL) Error Invalid Command Code

READ			
Request Syntax	*[R],[:],([addr_MSB]]...LSB])*		
Request example (no chks)	Read address: 0x0080: ASCII: #R:0080\n HEX: 23 *52 3A 30303830 0A*		
Request example (chks)	Read address: 0x0080 chks=crc8: ASCII: $R:0080*7B\n HEX: 24 52 3A 30303830 2A 3742 *0A*		
Argument description	*R*	Command code = 0x52	
	[addr_MSB]]...LSB]	Address, size: 1-4 bytes	
Response syntax (success)	*[R],[:],([data_MSB]]...LSB])*		
Argument description	*R*	Read success code = 0x52	
	[data_MSB]]...LSB]	Variable data group, size: 1-4 bytes	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_MAB	Error code (AL)	Error Missing Address Bytes
	E_FBR	Error code (AL)	Error Failed Bus Read

READ MULTIPLE			
Request Syntax	*[RM],[:],[length],([addr_MSB]]…LSB]))))*		
Request example (no chks)	Read Multiple addresses: 0x0080 and 0x010F ASCII: #RM:020080010F\n HEX: 23 524D 3A 3032 30303830 30313046 *0A*		
Request example (chks)	Read Multiple addresses: 0x0080 and 0x010F chks=crc8 ASCII: $RM:020080010F*6C\n HEX: 24 524D 3A 3032 30303830 30313046 2A 3643 *0A*		
Argument description	*RM*	Command code = 0x5253	
	length	Number of addresses in the message. The C_AL_RM_SIZE generic has a valid range between 0 and 255, [0x00-0xFF] and describes the maximum allowed number of addresses in a RM message. C_AL_RM_SIZE can be configured on synthesis, default = 32. (0: disables the RM command)	
	([addr_MSB]]…LSB]))))	Addresses, size: 1-4 bytes for each address	
Response syntax (success)	*[R],[:],[size],([data_MSB]]…LSB]))))*		
Argument description	*R*	Read success code = 0x52	
	size	Number of variable data groups in the message, *size=length*	
	([data_MSB]]…LSB]))))	Variable data group, size: 1-4 bytes for each data group	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_IBS	Error code (AL)	Error Invalid Block Size
	E_MAB	Error code (AL)	Error Missing Address Bytes
	E_FBR	Error code (AL)	Error Failed Bus Read

READ SUBSCRIPTION INFO			
Request Syntax	*[RSI],[:],[group_id]*		
Request example (no chks)	Read Subscription Info from group 1: ASCII: #RSI:01\n HEX: 23 *525349 3A 3031 0A*		
Request example (chks)	Read Subscription Info from group 1: chks=crc8 ASCII: $RSI:01*A8\n HEX: 24 *525349 3A 3031 2A 4138 0A*		
Argument description	*RSI*	Command code = 0x*525349*	
	group_id	Id of the subscription group to read data from. Valid range is controlled by the C_AL_SUBMNGR_SGID_SIZE generic. C_AL_SUBMNGR_SGID_SIZE ϵ [0:15]	
Response syntax (success)	*[R],[:],[rate]]],[size],([addr_MSB]]...LSB]))*		
Argument description	*R*	Success code = 0x52	
	rate	Current publish rate of the subscription group (relative to the publish base frequency of the subscription manager). Rate can consist of 1 to 4 bytes depending on the configuration of the C_AL_SUBGRP_RATE_BYTE_CNT generic. (default: 2) Rate bytes are transmitted MSB first : [rate_MSB]]...LSB]	
	size	Number of addresses in the message: *size ϵ [1:2^ C_AL_SUBGRP_ADDR_WIDTH] | ADDR_ C_AL_SUBGRP_ADDR_WIDTH=[2;5] (default: C_AL_SUBGRP_ADDR_WIDTH= 4)*	
	([addr_MSB]]...LSB]))	Addresses, size: 1-4 bytes for each address	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_MDB	Error code (AL)	Error Missing Data Bytes
	E_ISG	Error code (AL)	Error Invalid Subscription Group-id

WRITE			
Request Syntax	*[W],[:],[addr_MSB]]…LSB],[data_MSB]]…LSB]*		
Request example (no chks)	Write 0xDEADBEEF to address: 0x0084 (4 byte data, 2 byte address) ASCII: #W:0084DEADBEFF\n HEX: 23 57 3A 30303834 4445414442454646 *0A*		
Request example (chks)	Write 0xDEADBEEF to address: 0x0084 (4 byte data, 2 byte address): chks=crc8 ASCII: $W:0084DEADBEFF*27\n HEX: 24 57 3A 30303834 4445414442454646 2A 3237 0A		
Argument description	W	Command code = 0x57	
	[addr_MSB]]…LSB]	Address, size: 1-4 bytes	
	[data_MSB]]…LSB]	Data, size: 1-4 bytes	
Response syntax (success)	*[W]*		
Argument description	W	Success code = 0x57	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_MAB	Error code (AL)	Error Missing Address Bytes
	E_MDB	Error code (AL)	Error Missing Data Bytes
	E_FBW	Error code (AL)	Error Failed Bus Write

WRITE SUBSCRIPTION RATE			
Request Syntax	*[WSR],[:],[group_id],[rate]]]*		
Request example (no chks)	Write Subscription Rate, to group 1, rate=0x01F4 (rate size = 2 bytes) ASCII: #*WSR:0101F4\n* HEX: 23 575352 3A 3031 30314634 0A		
Request example (chks)	Write Subscription Rate, to group 1, rate=0x01F4 (rate size = 2 bytes) chks=crc8: ASCII: $*WSR:0101F4*07\n* HEX: 24 575352 3A 3031 30314634 2A 3037 0A		
Argument description	WSR	Command code = 0x575352	
	group_id	Id of the subscription group to write rate data to. Valid range is controlled by the C_AL_SUBMNGR_SGID_SIZE generic. C_AL_SUBMNGR_SGID_SIZE ϵ [0:15]	
	rate	Publish rate of the subscription group (relative to the publish base frequency of the subscription manager). Rate can consist of 1 to 4 bytes depending on the configuration of the C_AL_SUBGRP_RATE_BYTE_CNT generic. (default: 2) Rate bytes are transmitted MSB first : [rate_MSB]]…LSB]	
Response syntax (success)	*[W]*		
Argument description	W	Success code = 0x57	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_ISG	Error code (AL)	Error Invalid Subscription Group-id
	E_MDB	Error code (AL)	Error Missing Data Bytes

WRITE SUBSCRIPTION ADDRESSES			
Request Syntax	*[WSA],[:],[group_id],[length],([addr_MSB]]...LSB]))))*		
Request example (no chks)	Write Subscription Addresses, length = 0x02, Addr: 0x0080, 0x00C0: ASCII: #*WSA:02008000C0*\n HEX: 23 575341 3A 3032 30303830 30304330 0A		
Request example (chks)	Write Subscription Addresses, length = 0x02, Addr: 0x0080, 0x00C0 chks=crc8: ASCII: $*WSA:02008000C0*BC\n HEX: 24 575341 3A 3032 30303830 30304330 2A 4243 0A		
Argument description	*WSA*	Command code = 0x575341	
	group_id	Id of the subscription group to write rate data to. Valid range is controlled by the C_AL_SUBMNGR_SGID_SIZE generic. C_AL_SUBMNGR_SGID_SIZE ϵ [0:15]	
	length	Number of addresses in the message. Valid range is controlled by the C_AL_SUBGRP_ADDR_WIDTH generic. C_AL_SUBGRP_ADDR_WIDTH ϵ [4:32]	
	([addr_MSB]]...LSB]))))	Addresses, size: 1-4 bytes for each address	
Response syntax (success)	*[W]*		
Argument description	W	Success code = 0x57	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code
	E_ISG	Error code (AL)	Error Invalid Subscription Group-id
	E_MDB	Error code (AL)	Error Missing Data Bytes
	E_ILA	Error code (AL)	Error Invalid Length Argument
	E_MAB	Error code (AL)	Error Missing Address Bytes

ENABLE PUBLISH SERVICE		
Request Syntax	*[EPS]*	
Request example (no chks)	Enable Publish Service: ASCII: #EPS\n HEX: 23 455053 0A	
Request example (chks)	Enable Publish Service, chks=crc8: ASCII: $EPS*F4\n HEX: 24 455053 2A 4634 0A	
Argument description	*EPS*	Command code = 0x455053
Response syntax (success)	*[E]*	
Argument description	*E*	Success code = 0x45
Response syntax (error)	E_ICC	Error code (AL) Error Invalid Command Code
Publish syntax	*[P],[:],[group_id],[size],([data_MSB]]…LSB]))))*	
Argument description	*P*	Publish command/success code = 0x50
	group_id	Id of the subscription group generating the publish message. Valid range is controlled by the C_AL_SUBMNGR_SGID_SIZE generic. C_AL_SUBMNGR_SGID_SIZE ∈ [0:15]
	size	Number of variable data groups in the message: *size* will match the number of addresses programmed into the subscription group with the WSA command. Valid range is controlled by the C_AL_SUBGRP_ADDR_WIDTH generic. C_AL_SUBGRP_ADDR_WIDTH ∈ [4:32]
	([data_MSB]]…LSB]))))	Variable data group, size: 1-4 bytes for each data group

ENABLE PUBLISH SERVICE WITH CHECKSUM		
Request Syntax	*[EPC]*	
Request example (no chks)	Enable Publish service with Checksum: ASCII: #EPC\n HEX: 23 455043 0A	
Request example (chks)	Enable Publish service with Checksum, chks=crc8: ASCII: $EPC*84\n HEX: 24 455043 2A 3834 0A	
Argument description	*EPC*	Command code = 0x455043
Response syntax (success)	*[E]*	
Argument description	*E*	Success code = 0x45
Response syntax (error)	E_ICC	Error code (AL) Error Invalid Command Code
Publish syntax	*[P],[:],[group_id],[size],([data_MSB]]…LSB]))))*	
Argument description	*P*	Publish command/success code = 0x50
	group_id	Id of the subscription group generating the publish message. Valid range is controlled by the C_AL_SUBMNGR_SGID_SIZE generic. C_AL_SUBMNGR_SGID_SIZE ∈ [0:15]
	size	Number of variable data groups in the message: *size* will match the number of addresses programmed into the subscription group with the WSA command. Valid range is controlled by the C_AL_SUBGRP_ADDR_WIDTH generic. C_AL_SUBGRP_ADDR_WIDTH ∈ [4:32]
	([data_MSB]]…LSB]))))	Variable data group, size: 1-4 bytes for each data group

DISABLE PUBLISH SERVICE			
Request Syntax	*[DPS]*		
Request example (no chks)	Disable Publish Service: ASCII: #DPS\n HEX: 23 445053 0A		
Request example (chks)	Disable Publish Service, chks=crc8: ASCII: $EPC*9F\n HEX: 24 445053 2A 3946 0A		
Argument description	*DPS*	Command code = 0x445053	
Response syntax (success)	*[D]*		
Argument description	*D*	Success code = 0x44	
Response syntax (error)	E_ICC	Error code (AL)	Error Invalid Command Code

GAB-LINK UART VERSION

PHYSICAL LAYER

In the UART version of the GAB-Link stack the physical layer contains a UART with asynchronous Rx and Tx FIFO's. The reason for the FIFO's being asynchronous is to enable the Rx and Tx FSM's to operate at a clock frequency that is optimal for the selected BAUD rate and let the rest of the GAB-Link stack operate at a clock frequency that matches the user and application logic.

Figure 8 shows a block diagram of the physical layer and design of the UART.

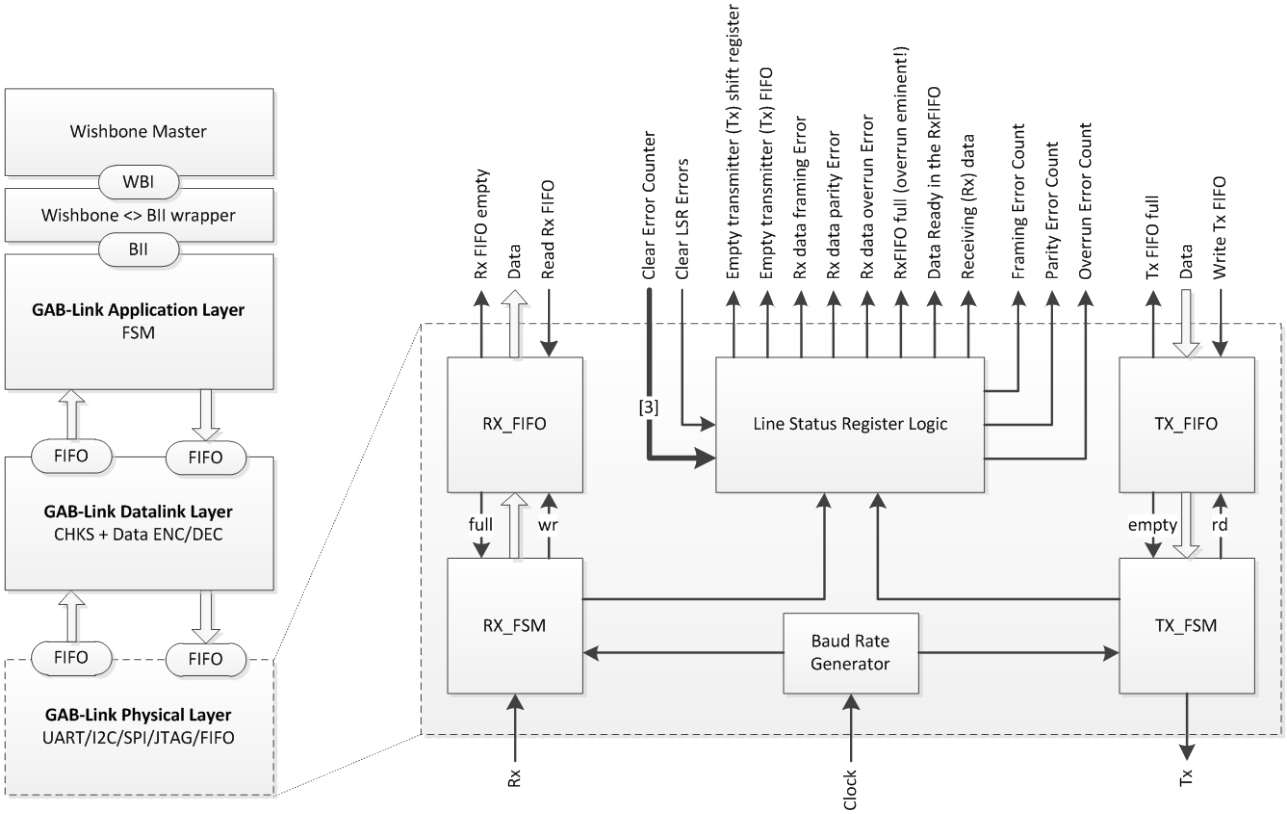

Figure 8: UART Physical Layer – GAB-Link

DATALINK LAYER

The purpose of the datalink layer is to ensure that only valid messages are passed on to the application layer. This is ensured by checking that a message is correctly formatted/framed and optionally if the checksum of a message is valid.

Figure 9 shows a block diagram of the datalink layer for the UART version of the GAB-Link stack. The datalink layer contains two main data paths the Rx and Tx paths and a secondary Error message path between the Rx and Tx paths.

The Tx path is the simplest of the major data paths, this contains a Tx FSM which retrieves data from the A and B Tx FIFO's. Data from the TxFIFO_A comes from the application layer and is prioritized over data from the TxFIFO_B which contains datalink layer error messages from the Rx path.
The Tx FSM writes data to the ASCII encoder unit which encodes data into ASCII hex codes, data marked with the control bit is not encoded but just passed directly to the Tx unit of the UART, this enables the possibility of transferring both control characters as well as binary data between the application and datalink layers using only 8 data bits and a single data/control bit.

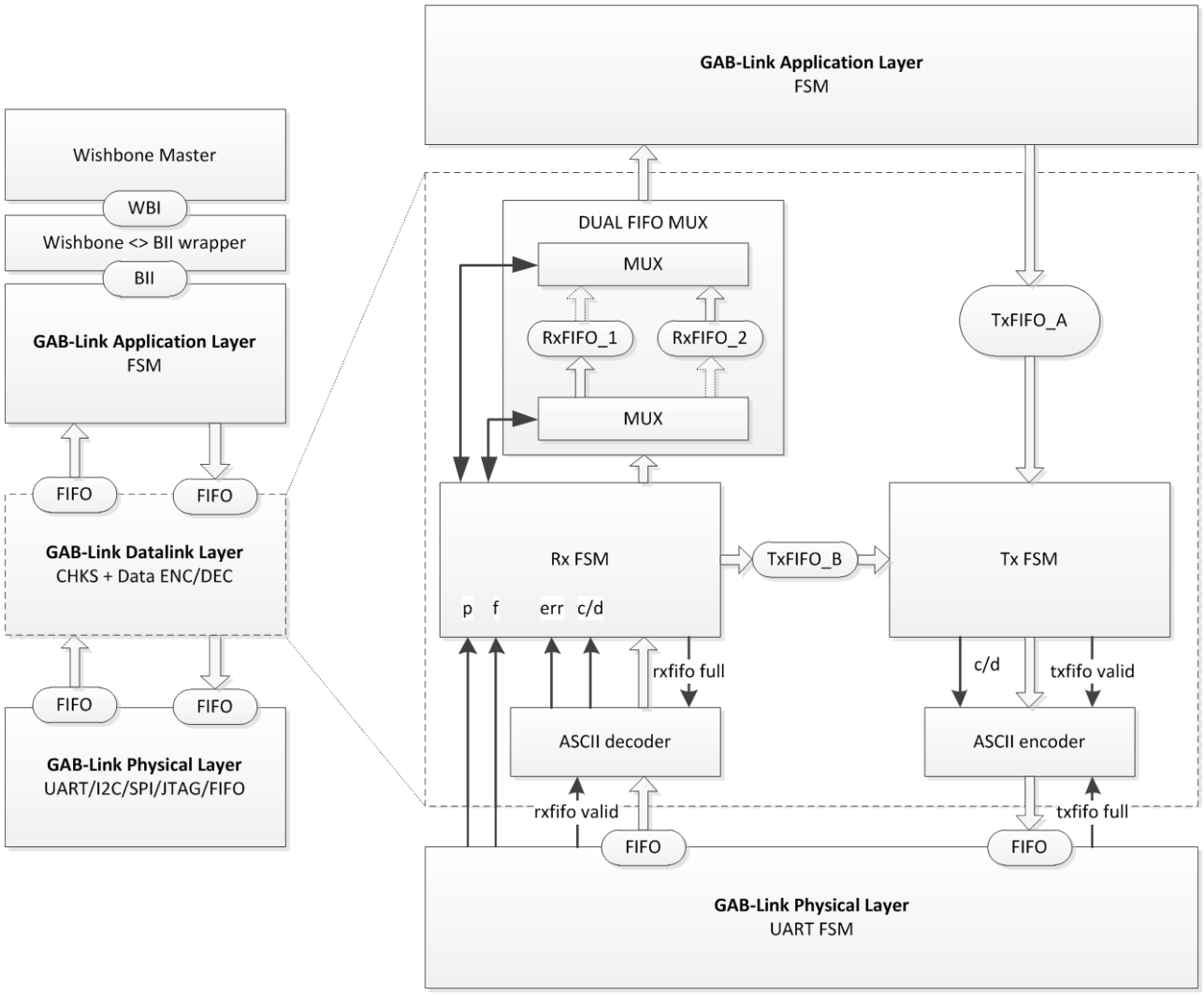

Figure 9: UART Datalink Layer – GAB-Link

The Rx path contains the Rx FSM which reads data (control characters as well as binary data) from the ASCII decoder. Valid binary data and header+tail control characters are passed to the Dual FIFO Mux which contains two Rx FIFO's (RxFIFO_1 and RxFIFO_2). When RxFIFO_1 is used for storing incoming data, RxFIFO_2 is being read by the application layer. When RxFIFO_2 is empty and a complete message is written (without errors) to RxFIFO_1, the roles as input/output Rx FIFO are exchanged, so the application layer can read the message in RxFIFO_1 and any incoming message can be written to RxFIFO_2.

If the header character dictates it, each payload cmd and data byte is used to calculate a CRC8 checksum which is validated against the checksum byte at the end of the message. Checksum data is not passed on to the application layer. The only knowledge of checksum the application layer gets is whether or not a request was sent with or without checksum, enabling it to request the datalink layer to transmit the response with or without checksum, simply by applying the correct header to the response message (i.e. the same header as that of the received request message).

If the datalink layer registers an error in the incoming message both the current input RxFIFO and the RxFIFO of the physical layer (the UART) are flushed and an error message is written to TxFIFO_B. Table 3 shows the error codes the UART datalink layer can transmit.

UART Datalink Layer Error Codes				
Response syntax (error)	0xF0	F_DPE	Error code (DL)	Data Parity Error
	0xF1	F_DFE	Error code (DL)	Data Framing Error
	0xF2	F_UBO	Error code (DL)	UART Buffer Overrun
	0xF3	F_MTL	Error code (DL)	Message Too Large/InFifo full
	0xF4	F_IMF	Error code (DL)	Invalid Message Formatting
	0xF5	F_MDB	Error code (DL)	Missing Data Byte
	0xF6	F_IMD	Error code (DL)	Invalid Data in Message
	0xF7	F_MCE	Error code (DL)	Message CHKS error

Table 3: UART Datalink Layer error codes

CRC8 CHECKSUM

The CRC checksum calculated by the CRC generator in the datalink layer is configured to generate a standard CRC-8-CCIT checksum. Several definitions of the CRC-8-CCIT checksum can be seen in Table 3.

CRC 8 CCIT	
Form	**Representation**
Polynomial	$x^8 + x^2 + x + 1$
Bit vector	100000111
Hex (normal)	0x07
Hex (reversed)	0xE0
Hex (reverse of reciprocal)	0x83

Table 4: CRC-8-CCIT checksum definitions

SMALL / LITE DATALINK LAYER

In order to ensure the GAB-Link stack can be implemented on very small FPGA's like the Spartan 3 XC3S50AN, the datalink layer can be implemented in a small (Lite mode) version, Figure 10 shows a block diagram of the Datalink layer configured in Lite mode.

The lite mode version of the datalink layer contains a reduced RxFSM which doesn't support checksum (CRC) verification, it doesn't handle parity or framing errors from the physical layer and it also doesn't handle errors from the ASCII decoder. It can only issue an error if an unsupported header character is transmitted as the beginning of a message.
All Rx and Tx FIFO's have been replaced with a data register that emulates the functionality of a FIFO but only can store a single entry.
These reductions ensure the datalink layer uses as little logic as possible while still retaining a minimum of functionality necessary for the GAB-Link stack.

If the Datalink layer is used in Lite mode incoming data messages will be forwarded directly to the application layer without any form of verification except that the header is valid, the content of a message, its size and tail are not checked. Use of the Lite mode is therefore NOT recommended for safety critical applications where an invalid message can't be allowed to accidentally alter the contents of a random memory register.

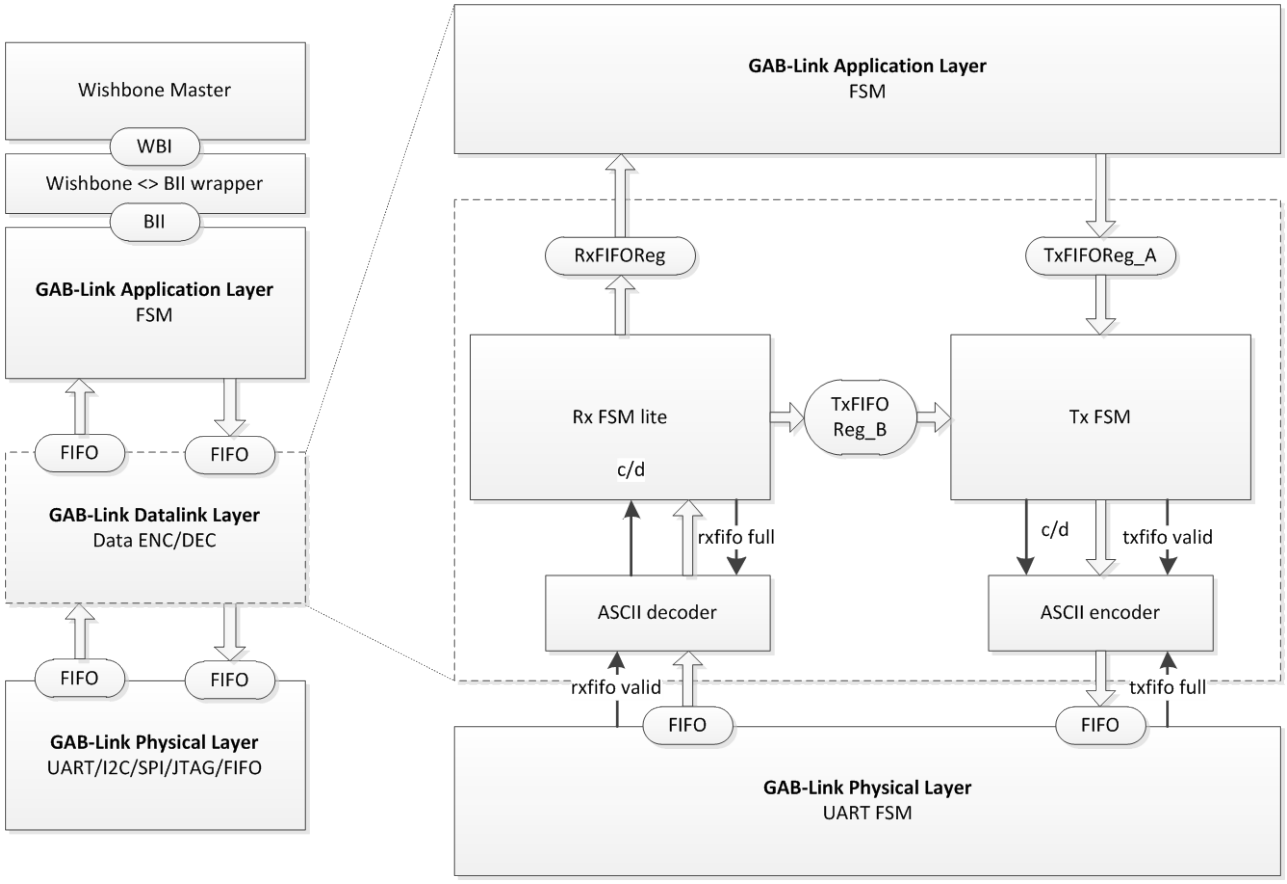

Figure 10: Lite UART Datalink Layer – GAB-Link

GAB-LINK APPLICATION LAYER

The application layer is implemented as a finite state machine, it starts in the *WAIT_HEAD* state where it waits for a publish request from the subscription group/manager (default priority 1) or the reception of a message header *#* or *$* (default priority 2).

The basic structure of the application layer finite state machine can be seen in Figure 11.

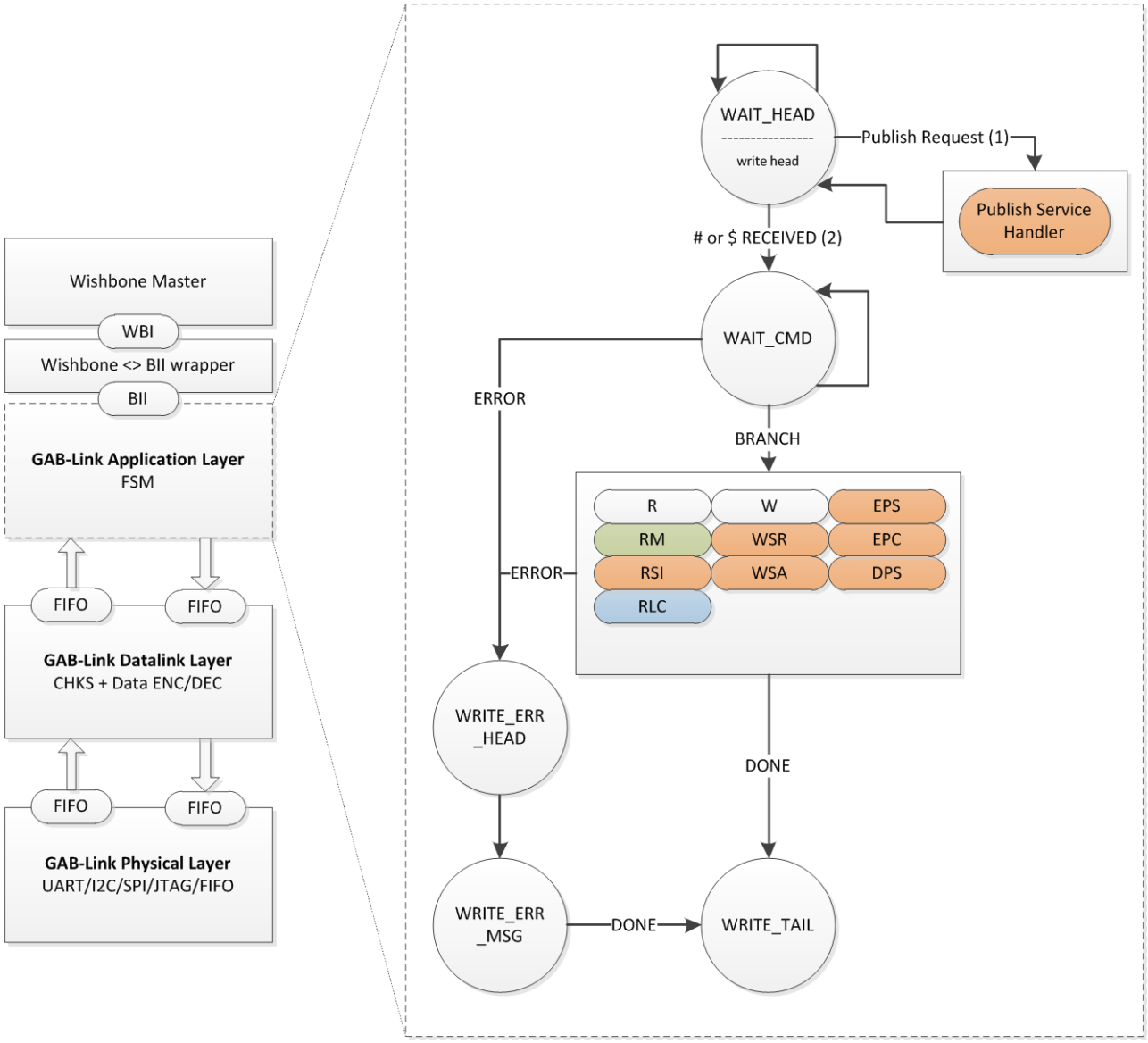

Figure 11: GAB-Link Application Layer

The orange, green and blue colored states are optional and can be removed from synthesis if desired in order to reduce the logic consumption of the GAB-Link stack.

If publishing is enabled the application layer also contains a subscription manager, holding one or more subscription groups, and a set of states implementing a Publish Service Handler, as well as states for activating and deactivating the publish service as well as configuring the individual subscription groups. The subscription groups can be configured with the addresses of data to be published and the

rate of which to publish them (relative to the base publish rate/frequency). The subscription groups each generates a publish request when the sync strobe counter matches the configured rate and if the publish service has been activated by the user, the Publish Service Handler states will be executed by the application layer FSM.

The base publish rate/frequency (sync strobe signal) is automatically generated by the subscription manager and its frequency can be configured prior to synthesis of the HDL for the GAB-Link. The publish rate (frequency) of each subscription group can be calculated by dividing the base publish frequency with the content of the subscription groups rate register:

$$publish\_freq = \frac{base\_publish\_freq}{subscription_{group_N}(publish\_rate)}$$

If a subscription groups publish rate is 0 or no addresses has been written to it (data_cnt = 0), publishing is disabled for that specific group.

Figure 12 shows a block diagram of the Subscription Group, it contains a sync strobe counter (sync_cnt), a rate register and data count (data_cnt) register as well as a small memory block (data0 to dataN-1) for holding the addresses of the data words to be published. The data_cnt register is configured with the number of data word addresses written to the data memory in order to enable the publish service to know how many addresses to read and hence dwords to publish.

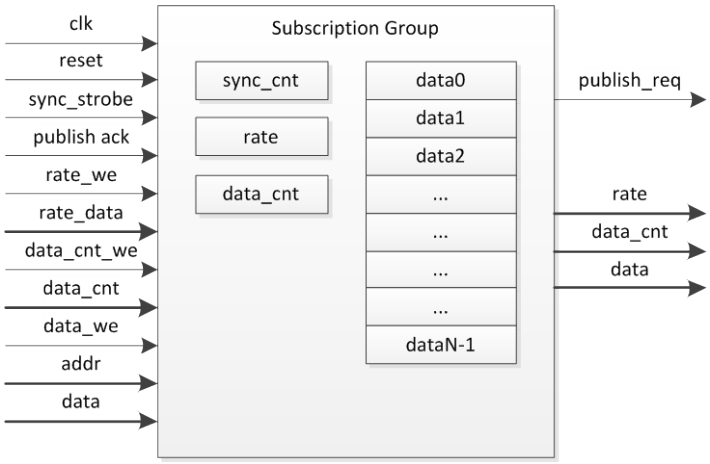

Figure 12: Subscription Group

When the header (# or $) of a message is received, the application layer FSM changes state to the *WAIT_CMD* state where it reads the next byte, verifies whether it is a valid command, if it is a command is branches to the request handler for the specific command.

The application layer FSM can handle the following requests:

- R : Read
- RM : Read Multiple
- RSI : Read Subscription info
- RLC : Read Link Configuration
- W : Write
- WSR : Write Subscription Rate
- WSA : Write Subscription Addresses
- EPS : Enable Publish Service (without checksum)
- EPC : Enable Publish service with Checksum
- DPS : Disable Publish Service

BUILDING THE UART GAB-LINK STACK

REQUIREMENTS:

Newest release of abl_vhdl_libs containing the following libraries:

- Checksum
- Clock_gen
- Encoder
- FIFO
- GAB_link
- Mod_m_counter
- Mux
- Ram
- Registers
- UART
- Utility
- Wishbone