

SHER-Bus Pre-specification

Warning

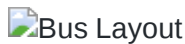
Content is not fixed and subject to change without notice!

SHER-Bus Stand for:

Systemwide Hub for Efficient Routing Bus

and

SHER-Bus Handles Extensive Resource Bridging, Unifying Systems



What is SHER-Bus?

It's a serial bus design to aggregate many low speed serial bus found in modern electronics to one/multiple high speed differential(s) pair(s). Low speed serial bus examples are SPI, I2C, CAN, PWM, PCM, etc. Gones are the day when you realize that you don't have enough i2c bus or UARTS.

This bus can also support more general usage like, humain-machine interface (keyboard, keypad, game controller, joysticks), audio (I2S, SPDIF), low-resolution video (monochrome oleds, spi tft lcd), motor control (stepper), General Purpose Input Output (GPIO), Battery managing (SOC, DOD, SOH, Series, Parallel, Charge/Discharge Current, Voltage, Temperatures) etc.

SHER-Bus is also great for general purpose communication (like UART, JASON, protobuf, modbus, etc) between controller.

Why SHER-Bus?

Sherbus is born of the frustration of working with the many, bloated, slow or proprietary communication protocol in embedded device. Sherbus is lightweight and easy to understand by anyone from the maker to the veteran designer. The protocol is modular (Only the Protocol layer [P] is mandatory). This modularity give the implementers leway to mold the communication to the product and not the otherwise. Implementers won't need to manage a complex stack for something simple (Like for example a full USB CDC class for a basic UART). The complexity of the software is designed to scale up with the complexity of the bus. In other word, easy task are easily done on simple bus. Otherwise, The more feature added, the more care (ex, Bus position identification, multiple lane, etc) need to be done to ensure the integrity of the bus. That meant that the bar of entry is very low, but the bus still powerfull enough to handle complex task when Implementers need it. This is the inovation SHER-Bus hold beacuse current protocols are either too complex for simple task (a USB stack can take most of the program space on most microcontroller even Tiny-USB) or too simple for complex task. SHER-Bus is both simple for simple task and ressource full for power-user.

Another point as why we need a protocol like SHER-Bus is that as much RISC-V helped open the CPU market to licence free IP for CPU, many System on chip require the use of proprietary IP for its connectivity. One goal of SHER-Bus is that one day a 100% free and open source SOC would hit the market. With the help of SHER-Bus and many more project like this, the open source community can achieve this objective.



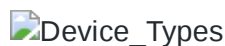
Bus design

Warning MLVDS is not a fixed choice and is subject to test and price comparison (MLVDS transceiver are expensive!)

The Bus is based on the M-LVDS (aka, TIA/EIA-899). Its design to support multipoint from the get go. The bus is wired-or (level high dominant). Up to 32 device (30 controller/bridge and 2 END bridge) can be connected to a single differential lane. 30 device can sound limited but the theoretical limit of i2c device on SHER-Bus is 22098!^[1] Multiple serial connection (like in a backplane) can be added to increase throughput. The clock is embedded into the datastream so no need to add a clock lane. An optional clock can be added to synchronise function like audio. It uses 8bit to 10bit encoding (or Manchester idk yet) on the physical layer to ensure DC balance and give a first layer of error checking.

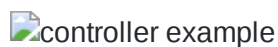
[^1]: 127 i2c device * 6 i2c master port on 1 bridge * 29 bridges (we need at minimum 1 controller) = 22098

Bus Devices



There is only 3 type of device that serve different function in the bus network.

First there is the controller that generate data packet for other to parse. They give the bus his function. For instance, the controller can give command to a bridge for him to read an i2c temperature sensor. The controller then interpret that data and then adjusting an spi DAC to output a value for the control of a Fan. They can also give command to other controller on the same bus. So in other word their job is to be the brain of the communication. They consist of mostly of Microcontroller, Embedded computer (Raspberry pi) or FPGA.



In second there is the bridge which is job is to close the gap between the new bus and the already existing protocol. They can have multitude of serial bus (up to 6 and 1 general control channel using [BPI]). They consist of mostly ASIC or FPGA. Although at the time of writing no ASIC or FPGA code as been manufactured/written. Microcontroller can also act as bridge. Bridge can be integrated into already existing design in die or in multi-die package. Following the design once reuse many, it help reduce redesign complexity and speeding up time to market. Bridge implementer need to be careful about licencing of existing bus as some are not free to implement.



In third, There is the End bridge which job is to connect multiple SHER-Bus together or other HIGH-SPEED bus (like USB, SDIO or Ethernet). They terminate the bus (where the termination resistor are) that's why they are a limit of 2 that can be connected on the bus. They are complex and not mandatory.

Packet achitecture

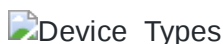
The packet architecture follow the same philosophy taken by the RISC-V cpu achitecture. Only one set of instruction is mandatory(P for SHER-Bus) and each set is labeled by a letter.(ex.: I M C A F D Q for risc-V CPU) or a word/achronim (ex.: Zicsr). What is different because of the nature of communication is that each layer is embeded into the payolad of layers under it. For example a Audio packet (A) is embeded into a Bus Protocol Identification [BPI] which is at his turn embeded into a stream package (S) which is at his turn embeded into a protocol package (S). Whe have structure like P(S(BPI(A))), but if that audio message is adressed to everyone on the bus we can remove the BPI layer and have message structured like this : P(S(A)). This greatly increase the flexibility of the bus. A reciver can perform and AND wise mask to the whole message to see if the message is of interest to him and discard those who aren't (ex.: an I2C brdge dont care about an audio(A) package but an I2S bridge do).



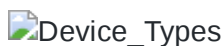
Protocol layer (P)

The protocol layer is the only mandatory layer of the specification. It handle the minimum to be a valid transaction. Implementer use this layer to send very low level message like a point to point UART-like communication [^2] , since adresssing is handeled in higher level only point to point or multidrop bus configuration is possible with using only this level. This is a feature because in many aplication you wouldn't want a heavy stack for something simple. This also grant implementer the freeddom to create custom protocol stack for application that arent covered by the existing stack.(ex.: SAE J1939 and CanOPEN are both stack that are built upon the unrestrictive nature of the CAN protocol, SHER-Bus trying to do the same but free with having a common stack help bridge implemter and bus implementer have a common ground to work with). the first byte is to tell if its a stadard packet or a custom one. A one(1) on the MSB of the first byte tell that the payload is a SER-Bus compliant message. Implementer can send custom message by setting the first byte to 0(0x00).

[^2]: only 2 Controller should be connected on the bus. A multipoint UART could be avallable in the application layer.



Network layer



Contol Messages (C)

Control message are use to change how the bus or a device react. For example an implementer can perform a device reset. It is also used by the [BPI] layer for Dynamic Adressing.

Interrupt Messages (I)

Interrupt Messages are designed to be as close as possible to computer interrupt. They can use the [BPI] but are more meant for Bus Wise attention. For example An ADC can send a Interupt to say that fresh data is ready to be read.

Bomerang Messages (B)

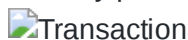
Bomerang Messages, like in their name suggest, are meant to come back to the sender. Upon reception of a (B) message the receiver use it to perform an action and then send back the same message with modification according to that action. The (B) message are an exception because they must support the Bus Position Identification messages, otherwise everyone on the bus would send back the message. Sending back the same message also ensure that it was received correctly and allow de synchronisation of transaction(the back and forth can happen with other message separating them). An example would be an I2C read on the bus. A controller could ask for read of W address X i2c slave on the Y I2C bus of Z bridge. The bridge would respond after performing the read that he received n bytes of data from W address X i2c slave on the Y I2C bus by sending back the same message but swapping the receiver/transceiver byte of the [BPI] and appending the readed data.

Stream Messages (S)

Stream message are for when data integrity is not important but a constant flow of data is. Stream message are for an audio stream for instance. Its not bad if 1 or 2 packet are lost its better to have a constant flow of packet. Stream Messages are better if they are on a separate bus as they are the lowest priority.

Bus Position Identification (BPI)

The Bus Position Identification give controller a way to talk with a specific device while other remain unchanged. Bridge must support BPI because they never know what bus they will be part of. Each device can have up to 7 sub-device. sub-device 0 is reserved for control or when sub-device are not used. Device can get an address of 3 way possible: Statically, Pseudo-Dynamically and Dynamically.



Static Addressing

Each device on the bus has a pre determined address that does not change. It is the responsibility of the Bus implementer to set a unique address to each device present on the bus. This mode of addressing is useful when the implementer knows the bus layout and it doesn't change like in a fully closed system with no exterior ports. To be documented.

Pseudo-Dynamic Addressing

Each device gets his address with using an external way. For example in a backplane each slot is labeled electronically(gpio or i2c eeprom) with a unique address. SHER-Bus device reads that address upon power up and uses it for communication. To be documented.

Dynamic Addressing

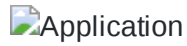
Each device gets his address asking the SHER-Bus using Control(C) message. To be documented.

High level Messages

Warning To be written

The high level message are functions that the bus can take. From the transport legacy protocol to a function based communication (Battery management, Data logging, Video, Audio) SHER-Bus can do it. Each

application is defined by a Application code that is given by the BUS community.(TO BE Deterined)



Transaction Example



In those example we can see two different protocol that can share the same bus. Firstm, we can see a controller try to read bytes off a i2c slave. In the second we can see a CAN packet. Its interesting than CAN message are sent using only the P layer. This show the SHER-Bus flexibility at handleling different aplication. SHER-Bus can use legacy adresssing method instead of it's BPI. This also show the power of having a mask on the whole packet instead of a specific region. The adresssing can change place or length in the packet but as long as the sender and reciver agree to it. The other on the bus simply ignore the message.