SHER-Bus Pre-specifcation

> [!IMPORTANT]
>
> Journée PMC du 8 decembre: pour des questions et commentaires utiliser la section discussions. Pour plus de détail sur la structure du PMC allez ici.

> [!Warning]
>
> Content is not fixed and suject 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?

SHER-Bus is a serial bus design to aggregate many low speed serial buses found in modern electronics to one/multiple high sppeed differential(s) pair(s) . Low speed serial bus examples include SPI, I2C, CAN, PWM, PCM, etc. Gone are the days when you realize that you don't have enough i2c buses or UARTS.

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

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

Why SHER-Bus?

Sherbus is born out of frustration from working with the many bloated, slow or proprietary communication protocols in embedded devices. Sherbus is lightweight and easy to understand by anyone from makers to veteran designers. The protocol is modular (Only the Protocol layer [P] is mandatory). This modularity gives implementers leeway to mold the communication to the product and not the other way around. 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

words, easy tasks are easily done on a simple bus. Otherwise, The more features are added (ex, Bus position identification,multiple lane, etc), the more care needs to be taken to ensure the integrity of the bus. That means that the barrier of entry is very low, but the bus is still powerful enough to handle complex tasks when Implementers need it to do so. This is the inovation SHER-Bus holds because current protocols are either too complex for simple tasks (a USB stack can take most of the progam space on most microcontrollers even Tiny-USB) or too simple for complex tasks. SHER-Bus is both simple for simple tasks and expandable for power-users.

Another point of why there's a need for a protocol like SHER-Bus is that as much as RISC-V has helped open the CPU market to lisence free IP for CPUs, many System-on-Chip designs (SoCs) require the use of prorietary IP for their 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 projects like this, the open source community can achive this objective.

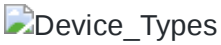
Bus design

> **Warning** MLVDS is not a fixed choice and is suject to test and price comparaison(MLVDS tranciver are expensive!)

The Bus is based on the M-LVDS(aka, TIA/EIA-899). It's design to support multipoint from the get go. The bus is wired-or (level high dominant). Up to 32 devices (30 controlers/bridge and 2 END bridges) can be connected to a single differential lane. 30 devices can sound limited but the theorical limit of i2c device on SHER-Bus is 22098![^1] Multiple serial connections (like in a backplane) can be added to increase throughput. The clock is embeded into the datastream so there's no need to add a clock lane. An optional clock can be added to sychronize functions 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.

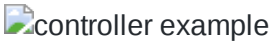
[^1]: 127 i2c device * 6 i2c master port on 1 bridge * 29 bridges (we need at minimum 1 controller) = 22098

Bus Devices



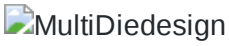
There are only 3 types of devices in the bus network, each serving a different function.

First there is the controller that generates data packets for others to parse. They give the bus its function. For instance, the controler can give commands to a bridge for it to read an i2c temperature sensor. The controller then interprets that data and adjusts a spi DAC to output a value for the control of a Fan. They can also give commands to other controllers on the same bus. So in other words their job is to be the brain of the communication. They consist of mostly of Microcontrollers, Embedded computers (Raspberry pi) or FPGAs.



Secondly there is the bridge whose job is to close the gap between the new bus and the already existing protocols. They can have mutitude of serial buses (up to 6 and 1 general control channel using [BPI]). They consist of mostly ASICs or FPGAs. Altough at the time of this writing no ASIC or FPGA code as been manufactured/written. Microcontrolers can also act as bridges. Briges can be integrated into already existing

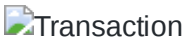
designs in die or in multi-die package. Folowing the design-once-reuse-everywhere rule, it helps reduce redesign complexity and speeding up time to market. Bridge implementers need to be careful about licensing of existing buses as some are not free to implement.



Thirdly, There is the End bridge whose job is to connect multiple SHER-Buses together or other HIGH-SPEED buses (like USB, SDIO or Ethernet). They terminate the bus (where the temination 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

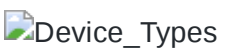
The packet architecture follows the same philosopy taken by the RISC-V cpu achitecture. Only one set of instruction is mandatory (P for SHER-Bus) and each set is labeled by a letter.(ex.: I M C A F D Q for risc-V CPU) or a word/achronim (ex.: Zicsr). What is different because of the nature of communication is that each layer is embedded into the payload of layers under it. For example a Audio packet (A) is embeded into a Bus Protocol Identification [BPI] which is in its turn embedded into a stream package (S) which in its turn embedded into a protocol package (P). We therefore 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 increases the flexibility of the bus. A reciver can perform an AND wise mask to the whole message to see if the message is of interest to it and discard those that aren't (ex.: an I2C bridge doesn't care about an audio(A) package but an I2S bridge migt).



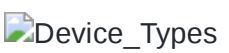
Protocol layer (P)

The protocol layer is the only mandatory layer of the specification. It handles the minimum payload to be a valid transaction. It consists of a single clockpulse followed by 32 bytes encoded in 8b/10b. Implementers use this layer to send very low level messages like a point-to-point UART-like communication [^2], since adressing is handeled at a higher level only point-to-point or multidrop bus configurations are possible while using only this level. This is a feature because in many applications you wouldn't want a heavy stack for something simple. This also grants implementers the freedom to create custom protocol stacks for applications that arent covered by the existing stacks. (Ex.: SAE J1939 and CanOPEN are both stacks that are built upon the unrestricitve nature of the CAN protocol, SHER-Bus is trying to do the same but free by having a common stack in order to help bridge implementers and bus implementers are starting from the same common point). The first byte indicates if it's a standard packet or a custom one. A one(1) on the MSB of the first byte indicates that the payload is a SER-Bus compliant message. Implementers can send custom messages by setting the first byte to 0(0x00).

[^2]: only 2 Controllers should be connected on the bus. A multipoint UART sould be availlable in the application layer.



Network layer



Contol Messages (C)

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

Interrupt Messages (I)

Interrupt Messages are designed to be as close as possible to traditional computer interrupts. 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)

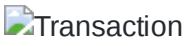
Bomerang Messages, like their name suggests, are meant to come back to the sender. Uppon reception of a (B) message the receiver uses it to perform an action and then send back the same message with modifcation according to that action. The (B) messages are an exception because they must support the Bus Position Identification messages, otherwise everyone on the bus would send back the message. Sending back the same message also ensures that it was received correctly and allow asynchronous coummunication -- i.e. per-transaction synchronisation (the back and forth can happen with other messages seperating 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 receiving 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/tranceiver byte of the [BPI] and appending the data that was read.

Stream Messages (S)

Stream messaged are for the use-case where data integrity is not important but a constant flow of data is. Stream messages are for an audio stream for instance. It's not bad if 1 or 2 packets are lost because it's better to have a constant flow of packets. 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 gives controlers a way to talk with a specific device while others remain unchanged. Bridges must support BPI because they never know what bus they will be part of. Each device can have up to 7 sub-devices. Sub-device 0 is reserved for control or when sub-devices are not used. Devices can get addresses in one of 3 ways: Statically, Pseudo-Dynamically and Dynamically.



Static Addressing

Each device on the bus has a pre determined address that doesn't change. It is the resposibility 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 dosen't change like in a fully closed system with no exterior ports. To be documented.

Pseudo-Dynamic Adressing

Each device gets its address by using an external mechanism. For eample in a backplane each slot is labeled electronically (gpio or i2c eeprom) with a unique address. SHER-Bus devices read that address upon

power up and use it for communication. To be documented.

Dynamic Adressing

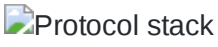
Each device gets its address by asking the SHER-Bus using a Control(C) message. To be documented.

High level Messages

> **Warning** To be written

High level messages are functions that the bus can take. From the transport legacy protocol to a function based ccommunication (Battery management, Data logging, Video, Audio) SHER-Bus can do it. Each application is defined by a Aplication code that is given by the BUS community.(TO BE Determined)

Transaction Example



In those examples we can see two different protocols that can share the same bus. First, we can see a controller try to read bytes off of an i2c slave. In the second we can see a CAN packet. It's interesting than CAN messages are sent using only the P layer. This shows SHER-Bus' flexibility at handling different aplications. SHER-Bus can use legacy addressing methods instead of its BPI. This also shows the power of having a mask on the whole packet instead of a specific region. The addressing can change place or length in the packet but as long as the sender and receiver agree to it then it'll work. The others on the bus simply ignore the message.