BT Car

# Overview

General scheme



# MCU DBG IF



General purpose – provide reliable (i.e. wired), stable and fast interface for debug logging and command processing without impact on basic (Bluetooth) interface.

As an interface USB through the external USB⬄FIFO converter (FT232HL) is selected. The converter allows seamless USB connection at speed ~8Mbytes/sec with very simple FIFO interface and doesn’t’ requires and USB specific functionality on MCU side. On a host side, converter uses vendor’s USB driver -[FTDI D2XX Driver](http://www.ftdichip.com/Drivers/D2XX.htm) .

On MCU side, the debug interface supports several independent streams:

* MCU DBG CMD/RESP

Bidirectional. Receives debug commands from host and send responses back to HOST.

* MCU DBG logging

Unidirectional. Sends log messages from MCU core functionality. I.e. messages generated locally by MCU.

* CSR DBG logging

Unidirectional. Sends log messages received from BT module (CSR) via UART interface.

The output stream is separated from each other on MCU level by packet header which contains stream ID and data length. Accordingly, packet header added to every packet being transferred through the USB IF to a HOST.



Figure . DBG IF stream header

On a HOST side, all data are received through a single USB interface (using FTDI driver API) and then split into different stream on application layer.

In the following TX & RX parts of MCU DBG interface are described in details.

### MCU DBG interfaces – TX part

#### General description

As already mentioned above, there are 3 TX interface – debug command responses, mcu debug logging and csr logging.



Figure . TX DBG interface data flow

As the interface supposed to be used in parallel, they can’t occupy MCU’s fifo port exclusively. Thus the dedicated buffer allocated in MCU’s memory for each interface (stream) to store output data temporary (Figure 2). All messages to be transferred through the interface are placed into the buffer allocated for that interface. The data might be filled as from application domain as well as from interrupt domain (ISR). The buffer might be unload in several ways depends on traffic load on these interface and overall MCU activity – from application domain and from FIFO related interrupts - TXE and Timer.

In the following the buffers’ structure and its specific is described in details.

#### TX buffer structure

There are three dedicated buffers:

* Command response - csr\_log\_buf[]
* MCU DBG logging - own\_log\_buf[]
* CSR DBG logging - cmd\_resp\_buf[]

All buffers are circular buffers with the same structure and size. Buffers’ size is selected as MAX\_UINT8 just in order to simplify circular addressing and arithmetic operations on the indices.

The buffers’ structure is in the following:

#define DBG\_BUFF\_LEN UINT8\_MAX

#define MEM\_SIGN\_LEN 4

#define BUFF\_GUARD\_LEN 4

typedef struct dbg\_buff\_tag {

uint8\_t uca\_guard\_s[BUFF\_GUARD\_LEN];

uint8\_t uca\_mem\_sign[MEM\_SIGN\_LEN];

uint8\_t uca\_data[DBG\_BUFF\_LEN];

uint8\_t uc\_rd\_idx;

uint8\_t uc\_wr\_idx;

uint8\_t uc\_bytes\_free\_sh;

uint8\_t uca\_guard\_e[BUFF\_GUARD\_LEN];

} DBG\_BUFF\_T;

uint8\_t uca\_guard\_x[BUFF\_GUARD\_LEN];

Just a memory guard area for easier memory corruption diagnostic.

Initialized at startup by constant 0xDEADBEEF.

uint8\_t uca\_mem\_sign[MEM\_SIGN\_LEN];

Memory area signature. Spefic for a particular object of parent type. Initialized at startup by a constant unique for that object instance.

uint8\_t uca\_data[DBG\_BUFF\_LEN];

Circular memory buffer. Non initialized. Read/write controlled by indices uc\_rd\_idx and uc\_wr\_idx.

uint8\_t uc\_rd\_idx;

Index for the next read operation. Initialized to 0. Read at index is allowed only if uc\_bytes\_free\_sh > 0. Every read by index operation increases the index and decreases uc\_bytes\_free\_sh by 1. The index is incremental by modulo DBG\_BUFF\_LEN.

uint8\_t uc\_wr\_idx;

Index for the next write operation. Initialized to 0. Write at index is allowed only if uc\_bytes\_free\_sh < DBG\_BUFF\_LEN - 2. Two bytes are preserved for overflow marking. Every write operation increases the index and decreases by uc\_bytes\_free\_sh by 1. The index is incremental by modulo DBG\_BUFF\_LEN.

uint8\_t uc\_bytes\_free\_sh;

Number of free bytes in buffer. Increased by 1 on read, decreased on write operations. The field is shared between application and ISR domains. I.e. read-modify-write operation in application area must be performed with disabled interrupts only.

#### TX buffer Initialization

Indices initialized to 0.

Memory signature initialized to the unique 4 byte constant.

Number of bytes free initialized to the buffer maximum value.

Buffer guard initialized to 0xDEADBEEF.

uint8\_t tx\_dbg\_buff\_init(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t \*puc\_sign);

#### TX buffers load

Writing to a debug buffer might be either from application domain (any function with the idle loop as a parent) or from interrupt service routine.

The application domain specific is that it might be hanging for some time awaiting until an event from interrupt unblocks it. Thus, in case of a TX buffer overflows during a writing operation the write function just polling buffer status until it will be unloaded from the TXE or Timer interrupt. Any shared variable accessed as read-modify-write must be decorated with interrupts disabling/enabling. Any non-atomic read operations (reading variables bigger than 1byte in size) must be decorated with interrupts disabling/enabling.

In opposite, writing to a buffer within ISR may not hang because of nested interrupts are not supported by the design (hanging in interrupts means MCU is dead). In case of a TX buffer overflows during an ISR execution, ISR writes the overflow mark to the buffer and terminates. Accordingly, TX buffers must always have free space preserved for the overflow mark. In case of several buffer’s overflow occurs in a row, the overflow mark should be update.

The writing from ISR might be executed in two different ways – bytewise and as a whole packet.

Bytewise writing is used when input data are not available all-at-once and can’t be read from a memory range, but are read from HW register or another serial source. The CSR log buffer is an example of bytewise writing.

##### *Write to buffer (non ISR version)*

void tx\_dbg\_buff\_wr\_non\_isr(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len);

void tx\_dbg\_buff\_wr\_cmd\_resp(uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len) {

tx\_dbg\_buff\_wr\_non\_isr(t\_cmd\_resp\_buff, puc\_in\_data, uc\_data\_len);

}

void tx\_dbg\_buff\_wr\_own\_log\_non\_isr();// wrappers for tx\_dbg\_buff\_wr\_non\_isr function

inline void

tx\_dbg\_buff\_wr\_non\_isr(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len) {

do {

if ((pt\_tx\_dbg->uc\_bytes\_free\_sh - OVERFLOW\_MARK\_SIZE) >= uc\_data\_len) {

// write to buffer. Update wr idx.

break;

}

else {

// Make sure or activate any buffer unload interrupt (by timer or by TXE)

}

} while (1);

DIS\_INT;

pt\_tx\_dbg->uc\_bytes\_free\_sh -= uc\_data\_len;

EN\_INT;

}

##### *Write to buffer (ISR version)*

// As the data incoming from CSR via the serial interface byte-by-byte, the function

// optimized for bytewise operation

void tx\_dbg\_buff\_wr\_csr\_log\_isr (uint8\_t \*puc\_in\_data) {

…

}

void tx\_dbg\_buff\_wr\_isr(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len);

void tx\_dbg\_buff\_wr\_own\_log\_isr (uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len) {

tx\_dbg\_buff\_wr\_isr(t\_own\_log\_buff, puc\_in\_data, uc\_data\_len);

}

inline uint8\_t

tx\_dbg\_buff\_wr\_isr(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t \*puc\_in\_data, uint8\_t uc\_data\_len) {

uint8\_t uc\_data\_written;

if (DATA+OVF\_MARK fit to buff) {

// Space OK. write to buffer

// Update buffer's vars

return NO\_ERR;

}

else {

return ERR\_OVF;

}

}

// In case of overflow, the ISR routine has to mark overflow in buffer for a whole packet

inline void tx\_dbg\_buff\_wr\_ovf(DBG\_BUFF\_T \*pt\_tx\_dbg, uint8\_t uc\_data\_len) {

if (OVF\_mark\_fit\_to\_buff){

// Write new OVF\_MARK

// Update buffer's vars

}else{

// Update existing OVF\_MARK

}

}

#### TX buffer unload

The buffers might be unloaded in the following ways:

* by idle loop. Unload data that currently are in buffer during idle loop execution.
* by timer interrupt. Called periodically in order to unload data from a buffer duringcommand execution or other application domain’s routine.
* by TXE interrupt. Triggered when the receiver is ready to accept new data.

**Unload by idle loop** - continuously check data presence in the buffers and if data are present, pushes a certain amount of data to the TX port. The function is called as one of the idle loop’s function. Because of others ISRs must be disabled during data writing to the I/O port, the amount of data to be written in one burst should be chosen considering overall system performance and should not blocks other interrupts significantly. The idle unload routine also resets the unload timer.

**Unload by timer**. The main purpose is to unload buffers during long app. domain operations with small traffic load. The buffer unloaded by the timer ISR that periodically triggered by HW. The timer’s ISR enabled at initialization time. The ISR stay enabled during normal operation unless buffer unloading by interrupt is enabled. In order to avoid excessive interrupt during the idle operation, unload by idle loop resets the timer’s counter.

**Unload by TXE interrupt**. The main purpose is to unload data from the buffers in case of high debug traffic load. The interrupt triggered by active (low) level on TXE# pin of external FIFO.

The ISR enabled then buffers’ fill level reach the certain enabling threshold - TXE\_EN\_THRS. Upon triggering, the ISR unloads certain amount of data and disable itself then buffer fill level reach the disabling threshold - TXE\_DIS\_THRS. As nested interrupts not supported by design, the amount of data to be unloaded at one ISR call should be chosen considering overall system performance allowing other ISRs to be served.

##### TX buffer unload (nonISR)

Non ISR version of unloading function called from the idle loop. It disables other unloading interrupts timer and TXE interrupts and fully unloads all TX buffers. Upon function completion the timer interrupt should be enabled again. The complete buffers unloading is feasible with the assumption that buffer’s unloading pace is much bigger than DBG data generating pace. Theoretical data rate on interfaces are in the following:

* CSR DBG interface maximum speed – 2Mbit/sec.
* Maximum USB speed – 64Mbit/sec
* Maximum FIFO data rate – ~8-16Mbit/sec (depend on implementation)

Data from a TX buffer prepended with buffer specific header (Figure 1) and then written to the USB FIFO. The buffer’s shared variables must be written with globally disabled interrupts. The number of operations executed with globally disabled interrupts must be as small as possible.

##### TX buffer unload (ISR)

In opposite to nonISR version, the function unloads certain amount of data from only one TX buffer. Amount of data should be small enough to not impact on other interrupts processing (lets choose 16 bytes per call). As the nested interrupts not supported by design, no precaution should be taken for shared variable access. In case of TX buffer overflow, the ISR must write overflow mark into the buffer and continue operation.

### MCU DBG interface – RX part





#### General flow

At start up program enters in idle loop and enables RXF interrupt (data in RX FIFO available). Upon new data arrival (2), HW triggers the interrupt. The ISR read the data byte from the FIFO port (2) and writes it to the global variable (3) shared between ISR and application domain. Then application disables itself at exit as the remaining part of incoming data will be read out from FIFO by the command from the application domain.

DBG command selector, which is running in idle loop, detects new command (4) and searches it within debug command table (5), then extracts address of command’s function and execute it (6). The command function reads out all required data directly from FIFO using RX FIFO polling (7).

Then, the command function returns to command selector which enables RXF interrupt and command selector returns to idle loop.

#### RX\_FIFO

RX\_FIFO is the connection between external FIFO and MCU. Consist of MCU’s HW port, configuration register and other HW related stuff.

#### RXF# interrupt service routine

RXF\_IRQ is the interrupt service routine (ISR) that triggered by active (low) level on RXF# during idle loop or in sleep mode. On interrupt trigger, the routine checks RXF# level and if it is active reads data from data port and store it in guc\_dbg\_cmd for further processing by DBG command executor. The ISR is supposed for the read of initial DBG command only, the rest of data are read by a particular command in polling mode.

The ISR disables itself on successful data read.

The ISR enabled just before entering while loop and after received DBG command processing.

The ISR is uninterruptible by any other ISRs.

#### Idle Loop (IL)

Is main loop that continuously executing during normal operation.

Idle loop might be interrupted by any interrupt.

IL continuously (or at wake up from an interrupt) checks debug command variable guc\_dbg\_cmd and if it not empty starts DBG command executor – proceed\_dbg\_command().

# CSR MCU IF



CSR MCU IF is the interface between MCU and Bluetooth (BT) module CSR1010.

The interface utilizes UART communication between BT module and MCU using internal HW on both sides. I.e. both MCU and CSR have HW support for UART interface.

### CSR side – TX part

In TX direction CSR MCU interface is used for several purposes:

* Send messages incoming from BT interface to MCU.
* Send debug log information from CSR module.
* Send responses on MCU command.

All type of data mentioned below generate by CSR User application (1), then prepended with dedicated header (2) and collected to the intermediate circular message buffer (3). This buffer unloaded by “TX complete” callback function (4) registered on UART initialization. The callback function triggers each time when HW finished send data over UART and is ready for a new data bunch. Upon trigger, the callback function writes data from message buffer to HW TX buffer (5) and exits. Upon data completely transferred over UART interface, CSR’s HW triggers the callback again. The callback function again reads data from message buffer to HW TX buffer and the sequence repeated until message buffer becomes empty. When all data from message buffer are transferred the callback function raises dedicated flag “TX empty” and exit. Then, next time, when CSR user application writes to the message buffer it checks the “TX empty” flag and if it is set, then initiate transfer by calling “TX complete” callback function.



#### TX Message buffer structure

typedef struct dbg\_buff\_tag {

uint16 usa\_guard\_s[BUFF\_GUARD\_LEN];

uint16 usa\_mem\_sign[MEM\_SIGN\_LEN];

uint16 usa\_data[TX\_MSG\_BUFF\_LEN];

uint16 us\_rd\_idx;

uint16 us\_wr\_idx;

uint16 us\_bytes\_free; // ??? Is really needed ???

uint16 us\_tx\_empty;

uint16 usa\_guard\_e[BUFF\_GUARD\_LEN];

} TX\_MSG\_BUFF\_T;

TX message buffer is a typical circular buffer with two indices ??? and one derivative variable free space ???. As the CSR is based on 16bit architecture (XAR) the native 16bit variables are used for indices and other buffer related variables. For the same reason data into the buffer are stored in so called packed form as an array of uint16.

uint16 usa\_guard\_s[BUFF\_GUARD\_LEN]

Just a memory guard area for easier memory corruption diagnostic.

Initialized at startup by constant 0xDEADBEEF.

uint16 usa\_mem\_sign[MEM\_SIGN\_LEN];

Memory area signature. The signature is specific for a particular object of that type. Initialized at startup by a constant unique for that object instance.

uint16 usa\_data[TX\_MSG\_BUFF\_LEN];

Circular memory buffer. Non initialized. Read/write operations controlled by indices us\_rd\_idx and us\_wr\_idx.

uint16 us\_rd\_idx

Index for the next read operation by TX complete callback. Initialized to 0. The TX complete callback may read data up to us\_wr\_idx-1. The index is incremental by module TX\_MSG\_BUFF\_LEN.

uint16 us\_wr\_idx;

Index for the next write operation. Initialized to 0. Write at index is allowed only if complete message fits to the buffer with some preserved space for possible overflow message. The index is incremental by module TX\_MSG\_BUFF\_LEN.

uint16 us\_tx\_empty

The flag that indicates that all data transferred from TX HW buffer and TX complete callback won’t be triggered anymore. Accordingly, if the flag is set, then the message writing routine must explicitly call TX complete callback on a new message write in order to initiate data transfer from TX message buffer to TX HW buffer. The TX complete callback function sets the flag if no more data available for transmitting (TX MSG buffer is empty) and clears it otherwise.

The flag initialized to 1 at CSR application startup.

#### TX MSG BUFFER overflow condition handling

Handling the message buffer overflow condition is very similar to one described for MCU DBG IF.

If a new message doesn’t fit to the buffer, the overflow mark is written to the buffer. If overflow mark doesn’t fit too it means that buffer already overflowed and overflow mark is already in the buffer. In that case LEN field of existing overflow mark updated.

Open question: How to guarantee atomic access to buffer indices from TX complete callback and message write buffer?

TODO: Check how TX complete callback function called. (Disasm? or CSR forum?)

# PWM control

# 

# ADDENDUM

## Links

<http://www.societyofrobots.com/actuators_servos.shtml#digitalanalogservos>

## Coding guide line

Rules of thumb

* Follow the existing style in files.

I.e. do not mix different coding styles within one file

CSR user app must follows the style define in application example or one defined for MCU coding style, but only one style allowed per file.

MCU FW uses elemental standard (C99) data types defined in <stdint.h>

The FW uses so called Hungarian notation, where data types code as variable’s prefix.

The following prefixes are in use:

int8\_t/char - c

uint8\_t - uc

int16\_t - s

uint16\_t - us

int32\_t - l

uint32\_t - ul

int64\_t - ll

uint64\_t - ull

The type of a variable is embedded in the name of the variable, i.e. all variable names have a prefix in front of them to denote the type. For instance, us\_CRClsb means that the variable “CRClsb” is of the type unsigned short integer.

There are 3 additional prefixes used in variable names:

p for pointers

g for global variables and

a for arrays

t for typedef and structures

e for enums

For example, gps\_cc is a global pointer variable to a short integers.

In addition there is on suffix for shared variables. Shared variables are variables that might accessed from main loop scope and from interrupt service routine and requires special handling in order to exclude simultaneous access from different domains.

The ordering is always [g][p][type][a]<\_name>[\_sh].

Defines, enums and typedef definition must be declared in CAPS in the same way as in AVR GNU library

typedef struct CLK\_struct

{

register8\_t CTRL; /\* Control Register \*/

register8\_t PSCTRL; /\* Prescaler Control Register \*/

register8\_t LOCK; /\* Lock register \*/

register8\_t RTCCTRL; /\* RTC Control Register \*/

} CLK\_t;

typedef enum GMP\_RANDALG\_enum

{

GMP\_RAND\_ALG\_DEFAULT = 0,

GMP\_RAND\_ALG\_LC = GMP\_RAND\_ALG\_DEFAULT /\* Linear congruential. \*/

} GMP\_RANDALG\_t;

Comments:

Both types C and C++ are allowed

Delimiters:

4 spaces. No Tabs.

Space between function name and its arguments.

Function’s opening bracket on next line

Sources srting len < 80 char. Exception are allowed in some cases.

uint8\_t proceed\_command (uint8\_t uc\_mcu\_cmd)

{

ACTION\_t \*pt\_action;

uint8\_t (\*pf\_func)();

uint8\_t uc\_act\_cmd;

uint8\_t uc\_ret\_cmd;

uint8\_t uc\_i;

switch (uc\_mcu\_cmd) {

case ACTION\_PING :

uc\_ret\_cmd = action\_ping();

break;

case ACTION\_SELF\_WRITE:

// Paranoic Flash write protection

for (uc\_i = 0; uc\_i < 200; uc\_i++) {

if (uc\_mcu\_cmd != ACTION\_SELF\_WRITE) {

return 0;

}

}

break;

case ACTION\_SELF\_READ:

uc\_ret\_cmd = action\_self\_read();

break;

default:

break; // break from switch

} // end of switch

}