RS485 Condor DC-DC Control Board Secondary MCU Communication Protocol:

Overview:

This document includes the current RS485 communication protocol between the Master (FPGA) and the Slave (DC-DC control board secondary MCU).

It also includes 2 appendices providing functions which can be used to convert the ADC temperature result to the value it represents and verify the CRC.

Communication Protocol

Baud Rate: 115200 (This baud rate can be easily changed)

Note: It may take a few microseconds for the Slave to respond. The current code for the DC-DC Control Secondary MCU sets a flag to respond when it receives the Master’s request and then transmits when flag is checked in the infinite loop and may first have to encode new data.

The communication is a two-step process:

1) The Master (FPGA) sends a data-request frame.

2) The Slave (DC-DC Control Board Secondary STM32) sends the data frames.

1. Data Request Frame:

The Master sends this frame in order to request data from the Slave.

The data request frame can be changed in the cfg.c file of the Secondary STM32 code.

It is currently 0xAA.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Request Frame from Master to Slave (Table 1) | | | | | | | | |
| Bytes:  Total: 1 | Bit 7  (MSB) | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 (LSB) |
| Byte 1: | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

1. Data Frames:

The Slave sends these frames in response to a data request from the Master.

The Slave sends all of the data in one transmission[[1]](#footnote-1).

An explanation of each of the data variables is attached after table 2.

The data frames are currently as follows:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Frames from Slave to Master (Table 2) | | | | | | | | |
| Bytes:  Total: 9 | Bit 7  (MSB) | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 (LSB) |
| Byte 1: | Temp.  bit 7 | Temp.  bit 6 | Temp.  bit 5 | Temp.  bit 4 | Temp.  bit 3 | Temp.  bit 2 | Temp.  bit 1 | Temp.  bit 0 (LSB) |
| Byte 2: | Vin  bit 11 (MSB) | Vin  bit 10 | Vin  bit 9 | Vin  bit 8 | Temp.  bit 11 (MSB) | Temp.  bit 10 | Temp.  bit 9 | Temp.[[2]](#footnote-2)  bit 8 |
| Byte 3: | Vin  bit 7 | Vin  bit 6 | Vin  bit 5 | Vin  bit 4 | Vin  bit 3 | Vin  bit 2 | Vin  bit 1 | Vin  bit 0 (LSB) |
| Byte 4: | Vout bit 7 | Vout bit 6 | Vout bit 5 | Vout bit 4 | Vout bit 3 | Vout bit 2 | Vout  bit 1 | Vout  bit 0 (LSB) |
| Byte 5: | Iin  bit 11 (MSB) | Iin  bit 10 | Iin  bit 9 | Iin  bit 8 | Vout  bit 11 (MSB) | Vout  bit 10 | Vout  bit 9 | Vout  bit 8 |
| Byte 6: | Iin  bit 7 | Iin  bit 6 | Iin  bit 5 | Iin  bit 4 | Iin  bit 3 | Iin  bit 2 | Iin  bit 1 | Iin  bit 0 (LSB) |
| Byte 7: | Iout bit 7 | Iout bit 6 | Iout bit 5 | Iout bit 4 | Iout bit 3 | Iout bit 2 | Iout  bit 1 | Iout  bit 0 (LSB) |
| Byte 8: | *OVP* | *OCP* | *OTP* | *VinP* | Iout  bit 11 (MSB) | Iout  bit 10 | Iout  bit 9 | Iout  bit 8 |
| Byte 9: | crc8 bit 7 (MSB) | crc8 bit 6 | crc8 bit 5 | crc8 bit 4 | crc8 bit 3 | crc8 bit 2 | crc8 bit 1 (LSB) | crc8 bit 0 |

Variable Description

Important! The FPGA will convert all of the ADC reads into the values they represent, except possibly for temperature.

Temp.[12:0]: The 12-bit result read from the ADC. The maximal value of all of the ADC temperature results read by this MCU. Has not been converted to temperature. Two possible functions for converting this value to temperature are included in appendix I. There is an optional mode on the secondary which first converts the 12-bit raw temperature value to an 8-bit temperature corresponding to actually Celsius temperature with an offset of -60°C.

Vin[12:0]: The 12-bit result read from the ADC representing Vin. Has not been converted from ADC result to voltage.

Vout[12:0]: The 12-bit result read from the ADC representing Vout. Has not been converted from ADC result to voltage.

Iin[12:0]: The 12-bit result read from the ADC representing Iin. Has not been converted from ADC result to current.

Iout[12:0]: The 12-bit result read from the ADC representing Iout. Has not been converted from ADC result to current.

*res*: Reserved bit.

crc8[8:0]: The 8-bit CRC of all of the previous bytes (Not including the CRC byte). The Master must also calculate the CRC of all of the previous bytes (Not including the CRC byte) and verify that they are the same. Otherwise, the data has been corrupted. A function for calculating CRC in the same way that the STM32 calculates it has been included in appendix II.

Appendix I

Appendix I includes two different C functions for converting Temp from the ADC value into 8 bits of temperature in degrees C offset so that the entire range fits in the 8 bits.

Both of the functions work only in the case that:

* The resistor in series with the NTC in the voltage divider is 10K Ohms.
* Vref = 3.3V
* Vcc = 3.3V
* The NTC is Horizon P.N. MIS-RN0002649

Both of these functions have been tested to make sure that they output the expected temperature but should be tested on the actual NTC and calibrated.

The Excel sheet used to develop both of these functions is in R:/Team/Tzvi/Condor/DCDC\_secondary/Used\_For\_Development/Development\_Calculations\_And\_Tests/Temperature\_lookup\_table.xlsx

FUNCTION 1:

The first method uses a lookup table. It has high accuracy between -40 degrees C and 145 degrees C.

|  |
| --- |
| Temperature Calculate using Lookup Table: NOT TESTED/CALIBRATED WITH ACTUAL NTC |
| #include <stdint.h>  #include <math.h>  #define TEMP\_OFFSET\_FOR\_RS485 213  #define MIN\_TEMP\_KELVIN 213  #define MAX\_TEMP\_KELVIN 468  // ADC breakpoints between 0 and 4095 for the temperature lookup table  const uint16\_t adc0 **=** 157**;**  const uint16\_t adc1 **=** 511**;**  const uint16\_t adc2 **=** 1022**;**  const uint16\_t adc3 **=** 2434**;**  const uint16\_t adc4 **=** 3020**;**  const uint16\_t adc5 **=** 3505**;**  const uint16\_t adc6 **=** 3699**;**  const uint16\_t adc7 **=** 3871**;**  const uint16\_t adc8 **=** 3954**;**  const uint16\_t adc9 **=** 4035**;**  // The temperature in degrees K for each ADC breakpoint  const uint16\_t temp0 **=** 233**;**  const uint16\_t temp1 **=** 255**;**  const uint16\_t temp2 **=** 272**;**  const uint16\_t temp3 **=** 308**;**  const uint16\_t temp4 **=** 327**;**  const uint16\_t temp5 **=** 352**;**  const uint16\_t temp6 **=** 369**;**  const uint16\_t temp7 **=** 395**;**  const uint16\_t temp8 **=** 418**;**  const uint16\_t temp9 **=** 468**;**  // The slopes of each linear range  const double slope1 **=** **(**double**)** **(**temp1 **-** temp0**)** **/** **(**double**)** **(**adc1 **-** adc0**);**  const double slope2 **=** **(**double**)** **(**temp2 **-** temp1**)** **/** **(**double**)** **(**adc2 **-** adc1**);**  const double slope3 **=** **(**double**)** **(**temp3 **-** temp2**)** **/** **(**double**)** **(**adc3 **-** adc2**);**  const double slope4 **=** **(**double**)** **(**temp4 **-** temp3**)** **/** **(**double**)** **(**adc4 **-** adc3**);**  const double slope5 **=** **(**double**)** **(**temp5 **-** temp4**)** **/** **(**double**)** **(**adc5 **-** adc4**);**  const double slope6 **=** **(**double**)** **(**temp6 **-** temp5**)** **/** **(**double**)** **(**adc6 **-** adc5**);**  const double slope7 **=** **(**double**)** **(**temp7 **-** temp6**)** **/** **(**double**)** **(**adc7 **-** adc6**);**  const double slope8 **=** **(**double**)** **(**temp8 **-** temp7**)** **/** **(**double**)** **(**adc8 **-** adc7**);**  const double slope9 **=** **(**double**)** **(**temp9 **-** temp8**)** **/** **(**double**)** **(**adc9 **-** adc8**);**  /\*\*  \* **@brief** Lookup Temperature Calculation  \*  \* **@param** uint16\_t adc\_val the 12 bit value read from the ADC  \* **@param** int8\_t calibration\_value the constant to add to the temperature so that it is calibrated. The temperature has not yet been calibrated  \* **@retval** uint8\_t the temperature in the range -60 to 195 degrees (C) in an 8 bit format (subtract 60 to get the temperature in degrees C)  \*  \* <BR>  \* \par<b>Description:</**b**><br>  \* Uses a lookup table to approximate the NTC curve as 9 different linear functions to convert the ADC value into a Kelvin temperature. **\n**  \* The Kelvin temperature is then offset to create the 8 bit return value. **\n**  \* This function is not always completely accurate. **\n**  \* Accuracy should be as follows: **\n**  \* Temperature (C) => **\n**  \* Less than -60 degrees: NOT A VALID INPUT (it will be rounded up to -60 degrees) **\n**  \* -60 to -40: Within 13 degrees **\n**  \* -40 to -18: Within 02 degrees **\n**  \* -18 to -01: Within 01 degrees **\n**  \* -01 to +35: Accurate **\n**  \* +35 to +54: Within 01 degrees **\n**  \* +54 to +79: Within 02 degrees **\n**  \* +79 to +96: Within 01 degrees **\n**  \* +96 to 122: Within 02 degrees **\n**  \* 122 to 145: Within 02 degrees **\n**  \* 145 to 195: Within 06 degrees **\n**  \* Higher than 195 degrees: NOT A VALID INPUT (it will be rounded down to 195 degrees) **\n**  \* <BR>  \*/  uint8\_t Lookup\_Temp\_Calc**(**uint16\_t adc\_val**,** int8\_t calibration\_value**)** **{**  uint16\_t temp **=** 0**;** // In degrees Kelvin    // Check which range the adc value is in and then calculate the temperature aproximating the range to a linear function.  **if** **(**adc\_val **<=** adc1**)** **{**  temp **=** round**(**slope1 **\*** **(**double**)** **(**adc\_val **-** adc0**)** **+** **(**double**)** temp0**);**  **}**  **else** **if** **((**adc\_val **>=** adc1**)** **&&** **(**adc\_val **<=** adc2**))** **{**  temp **=** round**(**slope2 **\*** **(**double**)** **(**adc\_val **-** adc1**)** **+** **(**double**)** temp1**);**  **}**  **else** **if** **((**adc\_val **>=** adc2**)** **&&** **(**adc\_val **<=** adc3**))** **{**  temp **=** round**(**slope3 **\*** **(**double**)** **(**adc\_val **-** adc2**)** **+** **(**double**)** temp2**);**  **}**  **else** **if** **((**adc\_val **>=** adc3**)** **&&** **(**adc\_val **<=** adc4**))** **{**  temp **=** round**(**slope4 **\*** **(**double**)** **(**adc\_val **-** adc3**)** **+** **(**double**)** temp3**);**  **}**  **else** **if** **((**adc\_val **>=** adc4**)** **&&** **(**adc\_val **<=** adc5**))** **{**  temp **=** round**(**slope5 **\*** **(**double**)** **(**adc\_val **-** adc4**)** **+** **(**double**)** temp4**);**  **}**  **else** **if** **((**adc\_val **>=** adc5**)** **&&** **(**adc\_val **<=** adc6**))** **{**  temp **=** round**(**slope6 **\*** **(**double**)** **(**adc\_val **-** adc5**)** **+** **(**double**)** temp5**);**  **}**  **else** **if** **((**adc\_val **>=** adc6**)** **&&** **(**adc\_val **<=** adc7**))** **{**  temp **=** round**(**slope7 **\*** **(**double**)** **(**adc\_val **-** adc6**)** **+** **(**double**)** temp6**);**  **}**  **else** **if** **((**adc\_val **>=** adc7**)** **&&** **(**adc\_val **<=** adc8**))** **{**  temp **=** round**(**slope8 **\*** **(**double**)** **(**adc\_val **-** adc7**)** **+** **(**double**)** temp7**);**  **}**  **else** **{**  temp **=** round**(**slope9 **\*** **(**double**)** **(**adc\_val **-** adc8**)** **+** **(**double**)** temp8**);**  **}**    temp **+=** calibration\_value**;**    // Verify that the temperature is in the -60 to 195 (or other) range  **if** **(**temp **>** MAX\_TEMP\_KELVIN**)** **{**  temp **=** MAX\_TEMP\_KELVIN**;**  **}**  **else** **if** **(**temp **<** MIN\_TEMP\_KELVIN**)** **{**  temp **=** MIN\_TEMP\_KELVIN**;**  **}**    uint8\_t offset\_temp **=** **(**uint8\_t**)** **(**temp **-** TEMP\_OFFSET\_FOR\_RS485**);** // Offset the temperature so that it takes up 8 bits.  **return** offset\_temp**;**  **}** |

FUNCTION 2:

The second method uses a Maclaurin series polynomial to calculate the temperature. It has high accuracy from -60 degrees C to 37 degrees C and above 37 degrees loses accuracy. The MatLab produced polynomial should be more accurate at the higher temperatures according to MatLab. If the user finds a mistake in this function, the accuracy may become greater than the lookup table accuracy.

|  |
| --- |
| Temperature Calculate using Polynomial: NOT TESTED/CALIBRATED WITH ACTUAL NTC |
| #include <stdint.h>  #include <math.h>  #define POLYNOMIAL\_TERMS 8  #define TEMP\_OFFSET\_FOR\_RS485 213  #define MIN\_TEMP\_KELVIN 213  #define MAX\_TEMP\_KELVIN 468  #define MAX\_EXPONENT 10  double single\_nomial**(**uint16\_t base**,** uint8\_t expn**,** double coefficient**);**  uint8\_t Polynomial\_Temp\_Calc**(**uint16\_t adc\_val**,** int8\_t calibration\_value**);**  // Polynomial coefficients:  // The order of the coefficients is: P(POLYNOMIAL\_TERMS) ... P6, P5, P4, P3, P2, P1  // The use will be: P(POLYNOMIAL\_TERMS) + P(POLYNOMIAL\_TERMS-1)\*ADC^1 + ... + P2\*ADC^(POLYNOMIAL\_TERMS-2) + P1\*ADC^(POLYNOMIAL\_TERMS-1)  // ie. The first coefficient is multiplied by the highest exponent and the last coefficient is multiplied by 1 (The base to the 0th power)  const double polynomial\_coefficients**[**POLYNOMIAL\_TERMS**]** **=** **{**2.043E+02**,** 2.605E-01**,** **-**5.876E-04**,** 7.448E-07**,** **-**5.033E-10**,** 1.834E-13**,** **-**3.394E-17**,** 2.504E-21**};**  /\*\*  \* **@brief** polynomial temperature calculation  \*  \* **@param** uint16\_t adc\_val, the adc read to convert into temperature.  \* **@param** int8\_t calibration\_value the constant to add to the temperature so that it is calibrated. The temperature has not yet been calibrated  \* **@retval** uint8\_t the temperature in the range -60 to 195 degrees (C) (or as specified) in an 8 bit format (subtract 60 (or as specified) to get the temperature in degrees C)  \*  \* <BR>  \* \par<b>Description:</**b**><br>  \* Uses a polynomial to approximate the NTC curve as a polynomial to convert the ADC value into a Kelvin temperature. **\n**  \* The Kelvin temperature is then offset to create the 8 bit return value. **\n**  \* This function is not always completely accurate. **\n**  \* Accuracy should be as follows: **\n**  \* Temperature (C) => **\n**  \* Less than -60 degrees: NOT A VALID INPUT (it will be rounded up to -60 degrees) **\n**  \* -60 to +37: Within 03 degrees **\n**  \* +37 to 195: Within 38 degrees (NOT ACCURATE AT ALL) **\n**  \* Higher than 195 degrees: NOT A VALID INPUT (it will be rounded down to 195 degrees) **\n**  \* <BR>  \*/  uint8\_t Polynomial\_Temp\_Calc**(**uint16\_t adc\_val**,** int8\_t calibration\_value**)** **{**  double curr\_polynomial\_sum **=** polynomial\_coefficients**[**0**];** // Set the first nomial to first\_coefficient\*x^0  **for** **(**int i **=** 1**;** i **<** POLYNOMIAL\_TERMS**;** i**++)** **{**  curr\_polynomial\_sum **+=** single\_nomial**(**adc\_val**,** i**,** polynomial\_coefficients**[**i**]);**  **}**    uint16\_t temp **=** round **(**curr\_polynomial\_sum**);**    temp **+=** calibration\_value**;**    // Verify that the temperature is in the -60 to 195 (or other) range  **if** **(**temp **>** MAX\_TEMP\_KELVIN**)** **{**  temp **=** MAX\_TEMP\_KELVIN**;**  **}**  **else** **if** **(**temp **<** MIN\_TEMP\_KELVIN**)** **{**  temp **=** MIN\_TEMP\_KELVIN**;**  **}**    uint8\_t offset\_temp **=** **(**uint8\_t**)** **(**temp **-** TEMP\_OFFSET\_FOR\_RS485**);** // Offset the temperature so that it takes up 8 bits.  **return** offset\_temp**;**  **}**  /\*\*  \* **@brief** single nomial  \*  \* **@param** uint16\_t base, the base of the power. The base should be 12 bits.  \* **@param** uint8\_t expn, the exponent of the power. expn should be less than equal to ten.  \* **@param** double coefficient, the coefficient multiplied by the power.  \* **@retval** double the coefficient times the base to the power of expn: coefficient\*base^expn  \*  \* <BR>  \* \par<b>Description:</**b**><br>  \* Receives a coefficient, a base, and an exponent. Returns coefficient\*base^expn **\n**  \* The user should verify that there is no overflow from the double return type. **\n**  \* To that end, base should not be more than 12 bits, expn should not be greater than the maximum allowed exponent.  \* <BR>  \*/  double single\_nomial**(**uint16\_t base**,** uint8\_t expn**,** double coefficient**)** **{**  base **=** **(**base **&** 0xFFF**);** // Verify that base is 12 bits.  **if** **(**expn **>** MAX\_EXPONENT**)** **{**  expn **=** MAX\_EXPONENT**;** // Verify that the exponent is not greater than the maximum  **}**  // Check edge case expn = 0.  // (If the base and the exponent are both 0, the function will return the coefficient even though it is not defined)  **if** **(**expn **==** 0**)** **{**  **return** coefficient**;**  **}**    double nomial **=** coefficient**;**  // Calculate the power  **for** **(**int i **=** 0**;** i **<** expn**;** i**++)** **{**  nomial **\*=** base**;**  **}**  **return** nomial**;**  **}** |

Appendix II

A C function for calculating the 8-bit CRC (calc\_crc\_8). The STM32 uses hardware to implement this function.

|  |
| --- |
| CRC functions: TESTED |
| #include <stdint.h>  #define POLY 0x8D // The binary polynomial coefficients used for calculating the CRC  #define MSB\_MASK 0x80 // The mask for the MSB of an 8 bit variable  uint8\_t calc\_crc\_8**(**uint8\_t**\*** bytes**,** uint8\_t num\_bytes**);**  uint8\_t crc\_8**(**uint8\_t initial\_crc**,** uint8\_t input\_data**);**  /\*\*  \* **@brief** 8 bit CRC of multiple bytes  \* **@param** uint8\_t\* bytes - the array of bytes of which to calculate the CRC  \* **@param** uint8\_t num\_bytes - the number of bytes in bytes[]  \* **@retval** uint8\_t the 8 bit CRC  \*  \* <BR>  \* \par<b>Description:</**b**><br>  \* Uses crc\_8 to calculate the total CRC of all of the input bytes.  \* <BR>  \*/  uint8\_t calc\_crc\_8**(**uint8\_t**\*** bytes**,** uint8\_t num\_bytes**)** **{**  uint8\_t total\_crc **=** 0**;**  **for** **(**int i **=** 0**;** i **<** num\_bytes**;** i**++)** **{**  total\_crc **=** crc\_8**(**total\_crc**,** bytes**[**i**]);**  **}**  **return** total\_crc**;**  **}**  /\*\*  \* **@brief** 8 bit CRC  \* **@param** uint8\_t initial\_crc - the CRC of the previous input, or 0 if it is the first input.  \* **@param** uint8\_t input\_data - the newest byte of which to calculate the CRC  \* **@retval** uint8\_t the 8 bit CRC  \*  \* <BR>  \* \par<b>Description:</**b**><br>  \* Receives the previous CRC and a new byte and calculates a total CRC.  \* <BR>  \*/  uint8\_t crc\_8**(**uint8\_t initial\_crc**,** uint8\_t input\_data**)** **{**  uint8\_t bindex **=** 0**;**  uint8\_t crc **=** 0**;**  // Initial XOR operation with the previous CRC value  crc **=** input\_data **^** initial\_crc**;**  // The CRC algorithm routine  **for** **(**bindex **=** 0**;** bindex **<** 8**;** bindex**++)** **{**  **if** **((**crc **&** MSB\_MASK**)** **>** 1**)** **{**  crc **=** **(**crc **<<** 1**)** **^** POLY**;**  **}**  **else** **{**  crc **=** **(**crc **<<** 1**);**  **}**  **}**  **return** crc**;**  **}** |

1. Note: The Secondary STM32 has been programmed for the possibility of adding different types of data transmissions depending on the data request frame. Code must be added to the Secondary STM32 in order to use this possibility. [↑](#footnote-ref-1)
2. Raw temp data uses 12 bits. If the temperature data is converted to an 8-bit number, Byte2 bits 0-3 will be set to 0.

   3 The first 4 bits of byte 8 will transmit status information in case the power supply shuts down to describe the reason for the shutdown. OVP is on output. VinP is for input voltage (too high or too low). [↑](#footnote-ref-2)