CDM Controller

Generated by Doxygen 1.8.7

Mon Oct 20 2014 08:23:55

Contents

Chapter 1

Universal Intelligent Sensor Controller

This is a description of the CDM Universal Intelligent Sensor Controller.

```
Preliminary Documentation
```

CDM Development Document

The Universal Sensor Controller provides a number of commonly required SCADA features which traditionally require several separate devices.

- Standardized communication format for analog and digital sensors
- Programmable sensing intervals
- Shared communication infrastructure
- Conversion of readings to standard units of measure
- Detection and reporting of value ranges and alarm conditions
- Globally unique identification of every sensor
- Programmable control outputs
- A variety of local and wide-area communication options

BEWARE of INCONSISTENCIES,

UNIMPLEMENTED

and

UNINTENDED FEATURES

- Hardware Design
  - Features List
  - Rev A Design
  - Rev A Errata
- Firmware Design
- Sensors
  - Laser Fluid Level Sensor
  - Ullage Volume Sensor
- Communications
  - Wired Serial
    - RS-232
    - ModBus
  - Wireless Local Area
    - Wi-Fi
    - Bluetooth
  - Wireless Wide Area
    - Cellular GSM Short Message
    - Cellular CDMA Short Message
    - Satellite
- User Interface

Chapter 2

Laser Sensor User Guide

Command Overview

Command Sentences are received from the serial port. They may originate with a user, installer, computer data collection link or manufacturing test station.

The same command structure is used by stored commands within the Sensor. These commands are automatically executed when triggered by an event such as a timer, contact closure, or control panel request.

Command Sentences consist of one or more individual commands (with their optional parameters) followed by a checksum. All command and response characters are printable ASCII and designed to be human readable and (somewhat) intuitive. No pure binary data communication is expected or allowed. Command sentences may be up to 80 characters in length. It is generally expected that command sentences will be separated by line separators such as ASCII CR, LF, etc. Sending a CR before every command may be required in some situations, such as half-duplex RS-485, to ensure that the receive buffer properly frames the sentence and does not fail the checksum unnecessarily.

Individual commands consist of a single prefix character with optional numeric parameters. Numeric values may be either decimal or hexadecimal. Multiple numeric parameters are separated by commas and terminated by a semi-colon.

Commands may cause the reporting of a particular value, the setting of an internal value or the testing of a particular condition.

Reporting. Various values that are measured, calculated or stored in the Sensor may be reported. This example reports the measured values of the laser distance and temperature. Note that all response values are reported in decimal engineering units with appropriate signs and decimal point.

Example Command	Example Result
`a?;t?`	`a0:32.500in;t:85.2F`

Setting. There are several internal operation modes that can be set remotely, as well as sensor calibrations and the real-time clock. The `"="` before the parameter indicates that a new value is to be stored. The command may have multiple parameters separated by commas to allow for more convenient and intuitive entry.

Example Command	Example Result
`T=12,34,56;T?;d?`	`T:13-06-13,12:34:56;d:+0.0mB`

Testing. Sensors may be used on multi-drop communication links (such as RS-485) or with broadcast protocols. Conditional commands allow Sensor units to be individually selected. Conditional commands are placed at the beginning of a command Sentence. If any conditional fails, the remaining command Sentence is discarded (ignored).

Example Command	Example Result

s∼156;s?;d?	s:156,d:+0.0mB

This would come from the Sensor with serial number 156 and no other unit would respond.

In general, the "∼" character is used to indicate a conditional. The parameter following the "∼" is compared to the value stored internally and if the values do not match the test fails and the remainder of the command Sentence is discarded.

There are two special cases for conditional commands. The first is the checksum which is placed at the end of the command Sentence. If the checksum is present and does not match the value computed internally, the entire command Sentence is discarded. Checksums are optional if commands are being echoed, since we assume that echoing is being used by a person manually entering commands.

The second special case is command Sentences containing Sequence numbers. The "S" command specifies a command sequence number which must NOT match the previous sequence number. This is intended for use with unreliable communication links which may duplicate packets or drop responses, causing the host to send retries. This ensures that command Sentences that should be executed only once (such as incrementing the hour for Daylight Saving Time) can be sent over unreliable data links and yield the expected results.

Notes About Checksums

To help ensure valid communication checksums are used at the end of every Command or Response Sentence. These checksums follow the general format used by the National Marine Electronics Association and should be familiar to anyone who has worked with a GPS unit.

The checksum is transmitted as an ASCII "∗" character followed by two ASCII hex characters representing the hexadecimal value of the eight-bit checksum.

All characters from the beginning of the Sentence up to and including the "∗" are summed.

Non-Volatile Operating Parameters

The table shows the parameters that may be set to configure the Sensor for particular applications, or that may be used in response Sentences. These parameters are stored in the Sensor and are preserved in the event of power loss.

Parameter	Description	Examples
eeAlias	Customer's Sensor Serial Number Alias	
eeTimer	Automatic reporting interval in 1/8 seconds	
eeRxSeq	Command Sentence sequence number	
eeTxSeq	Response Sentence sequence number	
eeShow	Automatic Reporting Command Sentence	
eeFormatTime	Bits controlling the way the Date and Time are reported	
eeFlags	Bits controlling the displayed data formats	

The following non-volatile items are used internally and are not generally available to the end user. This table is included only to provide a complete picture of the Sensor internal operation. These parameters are stored permanently during the manufacturing process.

Parameter	Description	Examples
eeSerial	Sensor Serial Number 156	
eeVersion	Sensor Firmware Version Number	1.1.44
eeDate	Sensor Manufacturing Date	11Jun2013
eeID	Sensor Model Number and description	Ullage Sensor
eeVcc	Core Internal Regulator Calibration	4970
eeBattScale	Scale factor for external supply voltage	604
eePresScale	Scale factor to convert differential pressures	40
eePresOffset	Offset to compute absolute pressures	9550
eeCount	Reset Count	5

Individual operating modes are controlled by bits in the eeFormatTime and eeFlags values. These operating parameters have a command that allows them to be set individually or collectively, or viewed as a whole. Some of the bits are dependent on other parameters to complete their meaning: for example, Setting a particular date-display Order will do nothing unless you also request that the date actually be displayed.

The internal names and meanings of individual bits are described in the next tables.

Time Format Bit	Description	Command	Meaning
fFormatTime	Include time in result. If the clock has not been set since power-on, this will be elapsed time.		
fFormatDate	Include date in result. If the date has not been set since power-on this will be an integer number of elapsed days.		
fFormatDelim	Include Delimiters `"-"`, `","` and `":"` if needed		
fFormatMDY	Display Month, Day, Year order		
fFormatDMY	Display Day, Month Year order		
fFormatDD	Display days only (suppress month and year)		
fFormatMMM	Display the month name as three characters		
fFormatMS	Include milli-seconds in the displayed time value		

This table contains several examples of Time Format Byte values and the results that can be expected in the output. The Time Format byte that we are describing here sets the defaule format used by the Response Sentences and the user interface display. These same byte values may also be used in the "T" Command to override the default behavior and get the desired format.

Time Format Byte	Command	Resulting Format
52	`z=FF005200`	`25Dec13`
02	`z=FF000200`	`121225`

0E	z=FF000E00	12-25-13
03	z=FF000300	121225201918
07	z=FF000700	13-12-25,20:19:18
01	z=FF000100	201918
05	z=FF000500	20:19:18
83	z=FF008300	131225201918.500
87	z=FF008700	13-12-25,20:19:18.500

The operating Mode Flag bits are defined in the following table.

Mode Flag Bit	Description	Command	Meaning
feeMetric	Report results using metric units	z=00030000	PSI
		z=00030001	mBar
		z=00030002	
		z=00030003	
feeAltUnit	Use alternate units of measure		
fee	(currently unused)		
feeUnits	Display units of measure after every value	z=00100000	Units Off
		z=00100010	Units On
feeVolts	Display analog voltage associated with every reading	z=00100000	Volts Off
		z=00100010	Volts On
feeCounts	Display raw analog value associated with every reading	z=00200000	Raw Off
		z=00200020	Raw On
feeNVpair	Include the parameter name character and an = at the beginning of every value reported	z=00400000	Name Off
		z=00400040	Name= On
feeEcho	Echo characters received from the serial port. This is intended for use with manually entered commands from a terminal program. Setting this also suppresses the checksum on the end of response Sentences.	z=00800000	Echo Off

		z=00800080	Echo On

Chapter 3

Control-Data Module Interface Specification

PCD-Bus Protocol Overview

The Power-Clock-Data Bus provides power and bi-directional data to multiple devices attached via parallel-connected conductors. All participating devices Listen on the bus; any device may also Talk to originate a message. Collisions are detected and only one Talker will complete the message - the other Talker(s) will simply retry.

All messages are received by all participating devices as a broadcast. The message may indicate one or more specific recipients; un-addressed devices will discard the message. Messages are unambiguously framed by an Idle Bus time interval and each message ends with a LRC byte. Malformed messages are discarded by all devices.

Devices implement the PCD-Bus protocol using only two timing parameters: T_p Minimum clock-pulse time (ensures bit reception by all participants), and T_h Maximum clock-high time (used as the End-of-Message Timeout).

Data bytes are transferred MSB first (Big-Endian) with one bit on each rising edge of the clock. There is no minimum bit time and any participating device may extend a bit (delay the Talker) if necessary.

Messages should be considered to be Datagrams - there is no handshaking between devices inherent in the PCD-Bus protocol. Messages may be Commands which contain requests for a Response. Responses will be sent as subsequent messages by the recipient(s). The Application Level should ensure that dropped or duplicate messages do not cause adverse effects. To this end, Command Sentences may contain a Message Sequence Number field which will detect and discard most duplicate Commands and Responses.

PCD-Bus Electrical Overview

The Power-Clock-Data bus provides power to remote devices to support either central or distributed sources and regulation.

The power portion consists of:

- common Ground for all devices,
- an unregulated battery source (6 - 24V),
- a 5V regulated source and
- a switched regulated auxiliary supply of implementation-dependent voltage.

Data transfer is performed over two open-drain parallel lines with active pullups. These are referred to as Clock (C) and Data (D). Signaling is done with nominal TTL (5 volt) levels.

Efforts are made in the design to tolerate electrical noise, distributed capacitance, extended cable lengths and poor rise-times on the Clock and Data signal lines. The data rates are kept intentionally low (by modern standards) to allow the use of less expensive or more robust industrial cabling and connectors.

With the use of appropriate connectors, the bus is intended to be hot-plugable, although the unregulated supply voltage will always be present on the bus. Sleeping, non-participating or non-present devices will not affect the use of the bus by other devices.

PCD-Bus Signaling States

Clock and Data lines are effectively open-drain with weak pullups.

Documentation Convention is (C) and (D) lines float high, (c) and (d) driven low.

Fully Idle bus state is (C) (D).

Any participating device(s) may become busy and block transmissions for an unlimited amount of time by asserting (d). The Busy Idle bus state is then (C) (d).

Any device may request to make a transmission on a Fully Idle or Busy Idle bus by asserting (c). Any busy participant(s) will eventually discover the transmission request and release their busy assertion (d). When all busy participant(s) are ready to receive the bus state will become (c) (D).

The Talking Device will detect the Ready bus state (c) (D). The Talker then sends consecutive bits as follows:

- Set next data bit (MSB first) as either (d) or (D).
- Release clock (C) and wait for it to be seen high.
  - Note: Recipients may assert (c) to delay the bus on a bit-by-bit basis.
- Verify Data. Bits are seen as either (c)(d) or (c)(D) by all participants at this point.
  - Verification failure indicates a bus collision. The Talker abandons the transmission immediately by releasing (D) and (C). The other Talker continues with his message.
- Wait for at least T_p Minimum Clock-Pulse Time.
- If the message has more bits to send assert (c) and wait T_p Minimum Clock-Pulse Time. Go back to the beginning of bit transmission to send the remaining bits.
- If the message (including LRCC) is complete, release (D) and (C). The device is no longer a Talker at this point. Wait for T_h Maximum Clock High Time, verifying that (C) stays high indicating that another Talker is not continuing to send bits.

This concludes the message transmission.

PCD-Bus Message Format

All PCD-Bus messages are formatted with a header, body and LRC character. The Header and Body are variable-length byte strings.

The Header consists of the Sender's Serial Number and an optional list of Recipient Serial Numbers with leading minus (`"-"`) signs. These Serial Numbers are all in Six-Bit format with leading zero suppression.

Diagnostics and direct data transfers between devices are implemented by including a comma (`","`) followed by a Six-Bit Logical Port Number. The Logical Port Number provides a shorthand notation for reading and writing data to and from common peripheral devices and memory structures. Defined values and examples are included later.

The Header is separated from the Body by a colon (`":"`) character.

The Body usually consists of a Command or Response Sentence.

The SixBit Character Set

Several message fields use SixBit characters (as opposed to 7- or 8-bit ASCII or hexadecimal) in order to provide a compact, visually recognizable format for binary data. The SixBit character set that we use is:

Hexadecimal Conversion

	0	1	2	3
0	**0**	**G**	**W**	**m**
1	**1**	**H**	**X**	**n**
2	**2**	**I**	**Y**	**o**
3	**3**	**J**	**Z**	**p**
4	**4**	**K**	**a**	**q**
5	**5**	**L**	**b**	**r**
6	**6**	**M**	**c**	**s**
7	**7**	**N**	**d**	**t**
8	**8**	**O**	**e**	**u**
9	**9**	**P**	**f**	**v**
A	**A**	**Q**	**g**	**w**
B	**B**	**R**	**h**	**z**
C	**C**	**S**	**i**	**y**
D	**D**	**T**	**j**	**z**
E	**E**	**U**	**k**	**$**
F	**F**	**V**	**l**	**_**

Decimal Conversion

	0	1	2	3	4	5	6
0	**0**	**A**	**K**	**U**	**e**	**o**	**y**
1	**1**	**B**	**L**	**V**	**f**	**p**	**z**
2	**2**	**C**	**M**	**W**	**g**	**q**	**$**
3	**3**	**D**	**N**	**X**	**h**	**r**	**_**
4	**4**	**E**	**O**	**Y**	**i**	**s**	
5	**5**	**F**	**P**	**Z**	**j**	**t**	
6	**6**	**G**	**Q**	**a**	**k**	**u**	
7	**7**	**H**	**R**	**b**	**l**	**v**	
8	**8**	**I**	**S**	**c**	**m**	**w**	
9	**9**	**J**	**T**	**d**	**n**	**x**	

Command and Response Sentences

Commands and Responses are transmitted in human-readable text messages. Each data item is a self-describing, variable-length string of characters. Non-printable characters are ignored; spaces may be included for documentation or clarity.

Data Items consist of a single-character identifier with optional index, followed by an operator and a list of zero or more values.

Commas or semicolons are used to separate multiple data items.

A Sentence consists of one or more Data Items followed by a Checksum.

The following characters may be used in Sentences:

Type	Allowed Characters
Identifiers	Letters ("A"-"Z" and "a"-"z") Digits ("0"-"9")
Delimiter	Comma (","), Semicolon (";")
Checksum	Asterisk ("*")
Operators	Colon (":"), Equal ("="), Question Mark ("?") and Tilde("~")
Values	Digits ("0"-"9") Letters ("A"-"Z" and "a"-"z") Dollar ("$") Underscore ("_") Plus ("+") Minus ("-") Decimal (".")
Decimal	Digits ("0"-"9")
Integer	Digits ("0"-"9") Plus ("+") Minus ("-")
Float	Digits ("0"-"9") Plus ("+") Minus ("-") Decimal (".")
Hexadecimal	Digits ("0"-"9") Letters ("A"-"F" or "a"-"f")
Sixbit	Digits ("0"-"9") Letters ("A"-"Z" and "a"-"z") Dollar ("$") Underscore ("_")
Units	Letters ("A"-"Z" and "a"-"z")
Quotes	Single-Quote ("'"), Double-Quote ('"'), Accent-Grave ("`")

Command Sentences

Command Sentences are used to Set, Test and Query data items using the following formats:

1. Set a data item value using the *identifier* = *value* format
2. Test a data item value using the *identifier* ~ *value* format
3. Request a response with the data item value using the *identifier* ? format

Response Sentences

Response Sentences report values for data items using the *identifier* : *value* format.

Defined Data Items

Identifier	Description	Set	Test	Query	Response
v	Version Number	v=010123	v~010123	v?	v:010123
S	Serial Number	S=123	S~123	S?	S:123
H	Hardware Model String	H=Laser		H?	H:Laser

h	Hardware Style Code	h=3		h?	h:3
I	Identification String	I=Sensor1		I?	I:Sensor1
E	Error Codes			En?	En:msg
F	Forward String	F0="Hello"			
s	Sequence Number			s?	s:1F
t0	Running Time since last reset			t0?	t0:123.456 t0:123.456s
t1	Total Running Time since manufacture			t1?	t1:12345 t1:12345s
t2	Time Date Offset	t=12345		t2?	t2:1234567 t2:1234567s
t	Time of Day			t?	t:131415 t:13:14:15
d	Date	d=140530		d?	d:140530
m	Mode Bits	m=01FAC		m?	m:01FAC
M	Memory Dump			M3?	M3:30-436↩ F6D6D00
T	Temperature			T?	T:80 T:80F T:25C
P	Pressure			P?	P:14.7 P:14.7psi P:1000mb
D	Distance			D?	D:1.234 D:40.1in D:1.234m
V	Volume			V?	V:12.3 V:12.3gal V:35.6l
a	Analog Input (voltage)			a0?	a0:3.456 a0:3.456V
A	Analog Input (current)			A0?	A0:3.456 A0:3.456A
R	Report Destination usually Phone Number	R1=2142323198		R1?	R1↩ :2142323198
r	Report Descriptor	R0="S?;a0?"		r0?	R0:"↩ S?0;a0?"
i	Intervals for Reporting	i0=300		i0?	i0:300
c	Conversion Polynomial	c0=		c0?	c0:
z	Diagnostics			z0?	*EEPROM Dump*

Defined Logical Ports

Logical Ports are used to transfer binary data blocks between devices on the PCD-Bus.

Frequently it is necessary to be able to move data originating at one device to a control structure or peripheral hardware on another device.

If a Logical Port Number is included in the header of a PCD-Bus message it indicates that the message body is binary data to be directed to the port of the Recipient unit(s).

Port	Name	Description
0	ReadE	Read EEPROM Bytes. Hexadecimal aa nn
1	WriteEE	Write EEPROM Byte(s). Hexadecimal aa dd dd ...
2	ReadF	Read Flash Words. Hexadecimal aa aa nn
3	WriteF	Write Flash Word(s). Hexadecimal aa aa dddd dddd dddd ...
4	ReadM	Read Memory Bytes. Hexadecimal aa nn
5	WriteM	Write Memory Byte(s). Hexadecimal aa aa dd dd ...
..		
8	ReadC	Read Communication Ring
9	WriteC	WriteCommunication Ring
A	ReadU	Read User Display Buffer
B	WriteU	Write User Display Buffer
C		
D	WriteL	Write LED Control Registers. Hexadecimal aa dd dd
E	ReadL	Read remote Mode and Flag bits.
F	WriteM	Write remote Mode and Flag bits. Hexadecimal dddd
..		
a	ReadA	Read Analog Input Channel. Hexadecimal aa nn
b	WriteA	Write Analog Output Channel(s). Hexadecimal aa dddd dddd...
c	ReadD	Read Digital Input Bits. Hexadecimal aa nn
d	WriteD	Write Digital Output Channel(s). Hexadecimal aa dddd
e	ReadT	Read 32-bit Timer. Hexadecimal aa
f	WriteT	Write 32-bit Timer. Hexadecimal aa dddddddd
..		
$	Reset	Initiate Remote firmware reset.
_	Result	Results from Port Read requests will be sent to the Sender's Port Number.

Chapter 4

CDM ModBus Communication Module

Electrical Interface

The CDM Modbus Module uses either

Standard	Wires	Mode	Duplex	Style
RS-232	TX, RX, GND	Single-Ended	Full	Point-to-Point
RS-422	TxA, TxB, RxA, RxB, GND	Differential	Full	Point-to-Multi-Point
RS-485	A, B, GND	Differential	Half	Point-to-Multi-Point

For Differential Mode connections, Bus Termination is internally jumper selected.

Serial Interface

Serial Communication parameters are

Parameter	Values
Baud Rate	1200 2400 4800 9600 19,200 57,600 115,200
Data Bits	7 8
Parity	One Odd Even
Bit Order	LSB first
Stop Bits	One

ModBus Protocol

ModBus protocol parameters

ModBus Parameter	Values
Role	Master Slave
Slave Address	1 - 247
Communication Mode	ASCII RTU
Error Detection	LRCC (in ASCII mode) CRC-16 (in RTU mode)
Message Framing	":" and CR/LF (in ASCII mode) 3.5 char time idle (in RTU mode)

	ASCII mode	RTU mode
Message Header	":"	3.5 char time idle
Character Set	"0" - "9", "A" - "F"	8-bit binary
Error Detection	LRCC	CRC-16
Message Trailer	":"	3.5 char time idle

For ASCII mode the LRCC character is

- Seven Bits
- Initialized to 0x00
- The sum of all bytes in the message NOT including the leading ":"
- Finalized by subtracting from zero

For RTU mode the CRC-16 is

- 16 bits
- Initialized to 0xFFFF
- Polynomial is 0xA001 (*NOTE* This is not the CCITT CRC-16 polynomial 0x1021)
- Computation bit order is LSB first
- Output byte order is LSB first

Modbus Overview

ModBus implements a strict Master - Slave relationship among devices.

Only one master is allowed on a bus. Up to 247 Slave devices may be addressed.

The Master sends a Request message and waits for a Response message from the selected Slave.

Slave Address 0x00 is the broadcast address. All Slaves will honor Broadcast Requests but none will respond.

Individually addressed Slaves will send Response messages.

Requests and Responses are ModBus messages transferred in either ASCII or RTU mode.

RTU messages are streamed binary, checked with parity and CRC, and bracketed by idle intervals.

ASCII messages are printable hexadecimal characters, checked with LRCC, and bracketed by ":" and CR/LF characters.

16-bit Register values are transferred MSB first (Big Endian).

Bits within a byte are numbered 0..7 from LSB to MSB. 16-bit values use bit numbers 0..15 from LSB to MSB.

When discussing ModBus register addresses there are four distinct address groups. These groups are in DECIMAL address ranges with 9999 registers in each group.

Register Group	Operations	Decimal Address Range	Message Address Range
Discrete Output Bits (referred to as "Coils")	Read/Write	00001-09999	0x0000-0x270F
Discrete Input Bits	Read Only	10001-19999	0x0000-0x270F
Digital Input Registers (16-bit)	Read Only	30001-39999	0x0000-0x270F
Digital Output/Holding Registers (16-bit)	Read/Write	40001-49999	0x0000-0x270F

```
All Bit and Register Addresses suffer from an "off by one" problem.
Documentation will refer to "Coil 1" but the message address is 0x0000.

The concept of the ZERO had not been fully embraced in 1978
when the ModBus Standard Accretion began.
```

Request and Response Messages

Function Code	Format	Description
1	: ss 01 aaaa nnnn cc CR LF	Read Coil Status Request
1	: ss 01 nn dd dd .. cc CR LF	Read Coil Status Response
2	: ss 02 aaaa nnnn cc CR LF	Read Input Status Request
2	: ss 02 nn dd dd .. cc CR LF	Read Input Status Response
3	: ss 03 aaaa nnnn cc CR LF	Read Holding Register Request
3	: ss 03 nn dd dd .. cc CR LF	Read Holding Register Response
4	: ss 04 aaaa nnnn cc CR LF	Read Input Registers Request
4	: ss 04 nn dd dd .. cc CR LF	Read Input Registers Response
5	: ss 05 aaaa nnnn cc CR LF	Force Single Coil Request
5	: ss 05 aaaa nnnn cc CR LF	Force Single Coil Response
6	: ss 06 aaaa nnnn cc CR LF	Write Single Register Request
6	: ss 06 aaaa nnnn cc CR LF	Write Single Register Response
15	: ss 0F aaaa nnnn bb dd dd .. cc CR LF	Force Multiple Coils Request
15	: ss 0F aaaa nnnn cc CR LF	Force Multiple Coils Response
16	: ss 10 aaaa nnnn bb dddd dddd .. cc CR LF	Write Multiple Registers Request
16	: ss 10 aaaa nnnn cc CR LF	Write Multiple Registers Response
17		Report Slave ID Request

17		Report Slave ID Response
22		Mask Write Register Request
22		Mask Write Register Response
23		Read / Write Registers Request
23		Read / Write Registers Response

Example Messages

Function 1 - Read Coil Status

The request message specifies the starting coil and quantity of coils to be read.

Example of a **Request** to read 10...22 (Coil 11 to 23) from slave device address 4:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x04	"0" "4"	Device 4
Function	0x01	"0" "1"	Read Coils
Starting Address Hi	0x00	"0" "0"	Start with Coil 11
Starting Address Lo	0x0A	"0" "A"	
Quantity of Coils Hi	0x00	"0" "0"	Read 13 Coils
Quantity of Coils Lo	0x0D	"0" "D"	
Error Check Lo	0xDD	"E" "4"	
Error Check Hi	0x98		
Trailer		CR LF	
Total Bytes	8	17	

The coil status response message is packed as one coil per bit of the data field. Status is indicated as: 1 is the value ON, and 0 is the value OFF. The LSB of the first data byte contains the coil addressed in the request. The other coils follow toward the high-order end of this byte and from low order to high order in subsequent bytes. If the returned coil quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeroes (toward the high-order end of the byte). The byte count field specifies the quantity of complete bytes of data.

Example of a **Response** to read 10...22 (Coil 11 to 23) from slave device address 4:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x04	"0" "4"	Device 4
Function	0x01	"0" "1"	Read Coils
Byte Count	0x02	"0" "2"	Data Bytes to follow
Data	0x0A	"0" "A"	Coils 11-18
Data	0x11	"1" "1"	Coils 19-23
Error Check Lo	0xB3	"D" "E"	
Error Check Hi	0x50		
Trailer		CR LF	
Total Bytes	7	15	

Function 2 - Read Input Status

Reads the ON/OFF status of discrete inputs in the slave.

The request message specifies the starting input and quantity of inputs to be read.

Example of a **Request** to read 10...22 (inputs 10011 to 10023) from slave device address 4:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x04	"0" "4"	Device 4
Function	0x02	"0" "2"	Read Inputs
Starting Address Hi	0x00	"0" "0"	Start with Input 10011
Starting Address Lo	0x0A	"0" "A"	
Quantity of Inputs Hi	0x00	"0" "0"	Read 13 Inputs
Quantity of Inputs Lo	0x0D	"0" "D"	
Error Check Lo	0x99	"E" "3"	
Error Check Hi	0x98		
Trailer		CR LF	
Total Bytes	8	17	

The input status response message is packed as one input per bit of the data field. Status is indicated as: 1 is the value ON, and 0 is the value OFF. The LSB of the first data byte contains the input addressed in the request. The other inputs follow toward the high-order end of this byte and from low order to high order in subsequent bytes. If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeroes (toward the high-order end of the byte). The byte count field specifies the quantity of complete bytes of data.

Example of a **Response** to read 10...22 (inputs 10011 to 10023) from slave device address 4:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x04	"0" "4"	Device 4
Function	0x01	"0" "2"	Read Inputs
Byte Count	0x02	"0" "2"	Data Bytes to follow
Data	0x0A	"0" "A"	Inputs 11-18
Data	0x11	"1" "1"	Inputs 19-23
Error Check Lo	0xB3	"D" "D"	
Error Check Hi	0x14		
Trailer		CR LF	
Total Bytes	7	15	

Function 3 - Read Holding Register

Read the binary contents of holding registers in the slave.

The request message specifies the starting register and quantity of registers to be read.

Example of a **Request** to read 0...1 (register 40001 to 40002) from slave device 1:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x01	"0" "1"	Device 1
Function	0x03	"0" "3"	Read Holding Register
Starting Address Hi	0x00	"0" "0"	Start with Register 40001
Starting Address Lo	0x00	"0" "0"	
Quantity of Inputs Hi	0x00	"0" "0"	Read 2 Registers
Quantity of Inputs Lo	0x02	"0" "2"	
Error Check Lo	0xC4	"F" "A"	

Error Check Hi	0x0B		
Trailer		CR LF	
Total Bytes	8	17	

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register the first byte contains the high-order bits, and the second contains the low-order bits.

Example of a **Response** to read 0...1 (register 40001 to 40002) from slave device 1:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x01	"0" "1"	Device 1
Function	0x03	"0" "3"	Read Inputs
Byte Count	0x04	"0" "4"	Data Bytes to follow
Data Hi	0x00	"0" "0"	Holding Register 40001
Data Lo	0x06	"0" "6"	
Data Hi	0x00	"0" "0"	Holding Register 40002
Data Lo	0x05	"0" "5"	
Error Check Lo	0xDA	"E" "D"	
Error Check Hi	0x31		
Trailer		CR LF	
Total Bytes	9	19	

Function 4 - Read Input Registers

Read the binary contents of input registers in the slave.

The request message specifies the starting register and quantity of registers to be read.

Example of a **Request** to read 0...1 (register 30001 to 30002) from slave device 1:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x01	"0" "1"	Device 1
Function	0x04	"0" "4"	Read Input Registers
Starting Address Hi	0x00	"0" "0"	Start with Register 30001
Starting Address Lo	0x00	"0" "0"	
Quantity of Inputs Hi	0x00	"0" "0"	Read 2 Registers
Quantity of Inputs Lo	0x02	"0" "2"	
Error Check Lo	0x71	"F" "9"	
Error Check Hi	0xCB		
Trailer		CR LF	
Total Bytes	8	17	

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register the first byte contains the high-order bits, and the second contains the low-order bits.

Example of a **Response** to read 0...1 (register 30001 to 30002) from slave device 1:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x01	"0" "1"	Device 1

Function	0x04	"0" "4"	Read Inputs Registers
Byte Count	0x04	"0" "4"	Data Bytes to follow
Data Hi	0x00	"0" "0"	Holding Register 30001
Data Lo	0x06	"0" "6"	
Data Hi	0x00	"0" "0"	Holding Register 30002
Data Lo	0x05	"0" "5"	
Error Check Lo	0xDB	"E" "C"	
Error Check Hi	0x86		
Trailer		CR LF	
Total Bytes	9	19	

Function 5 - Force Single Coil

Writes a single coil to either ON or OFF.

The request message specifies the coil reference to be written. Coils are addressed starting at zero-coil 1 is addressed as 0.

The requested ON / OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the coil to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the coil.

Example of a **Request** to write coil 173 ON in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x05	"0" "5"	Force Single Coil
Coil Address Hi	0x00	"0" "0"	Coil Number 173
Coil Address Lo	0xAC	"A" "C"	
Write Data Hi	0xFF	"F" "F"	Set Coil On
Write Data Lo	0x00	"0" "0"	
Error Check Lo	0x4E	"3" "F"	
Error Check Hi	0x8B		
Trailer		CR LF	
Total Bytes	8	17	

The normal response is an echo of the request, returned after the coil state has been written.

Example of a **Response** to write coil 173 ON in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x05	"0" "5"	Force Single Coil
Coil Address Hi	0x00	"0" "0"	Coil Number 173
Coil Address Lo	0xAC	"A" "C"	
Write Data Hi	0xFF	"F" "F"	Set Coil On
Write Data Lo	0x00	"0" "0"	
Error Check Lo	0x4E	"3" "F"	
Error Check Hi	0x8B		
Trailer		CR LF	

Total Bytes	8	17	

Function 6 - Write Single Register

Writes a value into a single holding register.

The request message specifies the register reference to be Written. The requested Write value is specified in the request data field.

Example of a **Request** to Write register 40002 to 00 03 hex in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x06	"0" "6"	Write Single Register
Register Address Hi	0x00	"0" "0"	Register Number 40002
Register Address Lo	0x01	"0" "1"	
Write Data Hi	0x00	"0" "0"	Value for Register Number 40002
Write Data Lo	0x03	"0" "3"	
Error Check Lo	0x9A	"E" "5"	
Error Check Hi	0x9B		
Trailer		CR LF	
Total Bytes	8	17	

The normal response is an echo of the request, returned after the register contents have been written.

Example of a **Response** to Write register 40002 to 00 03 hex in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x06	"0" "6"	Write Single Register
Register Address Hi	0x00	"0" "0"	Register Number 40002
Register Address Lo	0x01	"0" "1"	
Write Data Hi	0x00	"0" "0"	Value for Register Number 40002
Write Data Lo	0x03	"0" "3"	
Error Check Lo	0x9A	"E" "5"	
Error Check Hi	0x9B		
Trailer		CR LF	
Total Bytes	8	17	

Function 15 - Force Multiple Coils

Writes each coil in a sequence of coils to either ON or OFF.

The request message specifies the coil references to be written. Coils are addressed starting at zero - coil 1 is addressed as 0.

The requested ON / OFF states are specified by contents of the request data field. A logical 1 in a bit position of the field requests the corresponding coils to be ON. A logical 0 requests it to be OFF.

Example of a **Request** to write a series of ten coils starting at coil 20 in slave device 17.

The request data contents are two bytes: 0xCD 0x01 (1100 1101 0000 0001 binary). The binary bits correspond to the coils in the following way:

Bit↩:	1	1	0	0	1	1	0	1		0	0	0	0	0	0	0	1
Coil↩:	27	26	25	24	23	22	21	20		-	-	-	-	-	-	29	28

The first byte transmitted (CD hex) addresses coils 27 ... 20, with the least significant bit addressing the lowest coil (20) in this set.

The next byte transmitted (01 hex) addresses coils 29 and 28, with the least significant bit addressing the lowest coil (28) in this set.

Unused bits in the last data byte should be zero-filled.

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x0F	"0" "F"	Force Multiple Coils
Coil Address Hi	0x00	"0" "0"	Coil Number 20
Coil Address Lo	0x13	"1" "3"	
Number of Coils Hi	0x00	"0" "0"	Number of Coils
Number of Coils Lo	0x0A	"0" "A"	
Byte Count	0x02	"0" "2"	Data Bytes to follow
Write Data	0xCD	"C" "D"	Set Coils
Write Data	0x01	"0" "1"	Set Coils
Error Check Lo	0xBF	"F" "3"	
Error Check Hi	0x0B		
Trailer		CR LF	
Total Bytes	11	23	

The normal response returns the slave address, function code, starting address, and number of coils written.

Example of a **Response** to write a series of ten coils starting at coil 20 in slave device 17.

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x0F	"0" "F"	Force Multiple Coils
Coil Address Hi	0x00	"0" "0"	Coil Number 20
Coil Address Lo	0x13	"1" "3"	
Number of Coils Hi	0x00	"0" "0"	Number of Coils
Number of Coils Lo	0x0A	"0" "A"	
Error Check Lo	0x26	"C" "3"	
Error Check Hi	0x99		
Trailer		CR LF	
Total Bytes	8	17	

Function 16 - Write Multiple Registers

Writes values into a sequence of holding registers

The request message specifies the register references to be written. Registers are addressed starting at zero - register 1 is addressed as 0.

The requested write values are specified in the request data field. Data is packed as two bytes per register.

Example of a **Request** to write two registers starting at 40002 to 0x000A and 0x0102, in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x10	"1" "0"	Write Multiple Registers
Coil Address Hi	0x00	"0" "0"	Register Number 40002
Coil Address Lo	0x13	"0" "1"	
Quantity of Registers Hi	0x00	"0" "0"	Number of Registers
Quantity of Registers Lo	0x02	"0" "2"	
Byte Count	0x04	"0" "4"	Data Bytes to follow
Write Data Hi	0x00	"0" "0"	Data for Register 40002
Write Data Lo	0x0A	"0" "A"	
Write Data Hi	0x01	"0" "1"	Data for Register 40003
Write Data Lo	0x02	"0" "2"	
Error Check Lo	0xC6	"C" "B"	
Error Check Hi	0xF0		
Trailer		CR LF	
Total Bytes	13	27	

The normal response returns the slave address, function code, starting address, and quantity of registers written.

Example of a **Response** to write two registers starting at 40002 to 0x000A and 0x0102, in slave device 17:

Field Name	RTU (hex bytes)	ASCII (characters)	Meaning
Header		":" (Colon)	
Slave Address	0x11	"1" "1"	Device 17
Function	0x10	"1" "0"	Write Multiple Registers
Coil Address Hi	0x00	"0" "0"	Register Number 40002
Coil Address Lo	0x13	"0" "1"	
Quantity of Registers Hi	0x00	"0" "0"	Number of Registers
Quantity of Registers Lo	0x02	"0" "2"	
Error Check Lo	0x12	"D" "C"	
Error Check Hi	0x98		
Trailer		CR LF	
Total Bytes	8	17	

Function 17 - Report Slave ID

Function 22 - Mask Write Register

Function 23 - Read / Write Registers

CDM Command and Response Sentences

Chapter 5

TxAM Advanced Pump Controller

Engineering Notes

These notes refer to Advanced Pump Control Rev C (week 9 of 2014).

Physical

- The APC is a 3.8 inch by 4 inch double-sided, conformal-coated circuit board
- The 2x16 LCD Display is a centrally located module attached with 0.5 inch nylon stand-offs
- The five-button membrane keypad is connected via an 8-pin header
- Rocker switch SW1 is power on/off

Connector Pin-Outs

- Two 0.25 inch right-angle male quick-disconnects are used for power input
  - S1 - Battery Positive
  - S2 - Battery Negative
- Two 0.25 inch right-angle male quick-disconnects are used for motor drive power output
  - S3 - Motor Plus
  - S4 - Motor Minus
- Shrouded 3-pin right-angle header is used for a remote temperature sensor
  - P1-1 - +5V
  - P1-2 - Temp
  - P1-3 - GND
- RJ-45 Right angle female connector is used for RS-232 serial control and optional power
  - P2-1 - Vpp / -MCLR (Processor Reset)
  - P2-2 - Vcc
  - P2-3 - GND
  - P2-4 -

  - P2-5 -
  - P2-6 - +12V Power Out (or in - see JP1)
  - P2-7 - TxD (an RS-232 output signal)
  - P2-8 - RxD (an RS-232 input signal)

- 3-pin SIP 0.1" connector (**DNI**) for external +12V Power Option. Pins 2 and 3 shorted on the board. Semantics unclear.
  - JP1-1 - (label AIN)
  - JP1-2 -
  - JP1-3 - (label 12V, hard wired trace)

- 16-pin SIP 0.1" connector is used for the LCD display module NHD-0216KZW-AB5
  - J1-1 - GND
  - J1-2 - Vdd +5V
  - J1-3 - N/C
  - J1-4 - Register Select 0=Command 1=Data
  - J1-5 - Read/Write 0=Write 1=Read
  - J1-6 - Enable (on falling edge)
  - J1-7 - DB0
  - J1-8 - DB1
  - J1-9 - DB2
  - J1-10 - DB3
  - J1-11 - DB4
  - J1-12 - DB5
  - J1-13 - DB6
  - J1-14 - DB7
  - J1-15 - N/C
  - J1-16 - N/C

- 8-pin 0.1" header is used for the 5-button membrane keypad. When pressed ∼100ohm to ground.
  - J2-1 - LED+
  - J2-2 - GND
  - J2-3 - Button 1 (START/PAUSE)
  - J2-4 - Button 2 (TEST/PRIME)
  - J2-5 - Button 3 (UP)
  - J2-6 - Button 4 (DOWN)
  - J2-7 - Button 5 (ENTER)
  - J2-8 - GND

- 5-pin 0.1" header (**DNI**) is used for ICSP
  - J3-1 - Vpp / -MCLR
  - J3-2 - Vcc
  - J3-3 - GND
  - J3-4 - PGC

  - J3-5 - PGD
- Button-cell battery holder is used for on-board clock battery
  - BT1
- Fuse holder for Automotive Blade-type 15A fuse
  - F10 -

Principal Bill of Materials

Top Side

Ref Des	Description	Function
S1-4	0.5" RA Quick Disconnect	Power Connection
P1	0.1" Shrouded RA Header	Temperature Probe
P2	RJ-45F RA	RS-232 Serial Comm
J1	Conn SIP-16 0.1"	LCD display module
J2	Header SIP-8 0.1"	Keypad
J3	Header SIP-5 0.1" **DNI**	ICSP
JP1	0.1" 3-pin header **DNI**	Option- External power from P2
BT1	Coin cell holder	Clock Battery
C100	Cap 1000uF 35V Elect	12V Power Input Filter
SW1	Switch Rocker RA	Power switch
F10	Blade Fuse 15A	Primary 12V Fuse
D100	TVS V470	Surge Suppressor
R100	Res 0.01 ohm 5W	Motor current shunt
DS1	**DNI**	
Q200	**DNI**	

Bottom Side

Ref Des	Description	Function
C31	Cap 10uF Tant	Vcc Filter
D1	TPS30M60SG Schottky 60V 30A	Inductive Clamp Diode
D4		Supply Polarity Protection
Q1	IRFS3006 MOSFET 60V 240A N-Channel	High-Side Motor Switch
Q2		
Q3		Vcc Regulator +5V
U1		
U2		
U3		
U4	Max202 2 TX, 2 RX	RS-232 Interface
U5	PIC18F45K22 32K Flash, 1536 SRAM, 256 EEPROM, 36 I/O, 10-bit ADC, 2 USART	Microcontroller

U6	IR2118S	High-side MOSFET Driver
Y1		32KHz Crystal

ModBus Interface Module

Engineering Notes

The ModBus Interface Module is an electrical adapter that connects the serial interface (RJ-45 connector) on the Advanced Pump Controller to a wired Local Network.

The ModBus Module is a sealed (potted) controller measuring approximately 3" x 4" x 3/4".

Connection to the Pump Controller uses a Straight-Through Cable

- P1 RJ-45 Female connector for serial connection to Advanced Pump Controller
  - P1-1 - Vpp / -MCLR (Processor Reset)
  - P1-2 - Vcc (+5V)
  - P1-3 - GND
  - P1-4 - (probably PGC to allow ICSP)
  - P1-5 - (probably PGD to allow ICSP)
  - P1-6 - +12V Sensor Power for 4-20mA Channels
  - P1-7 - RxD (an RS-232 input signal)
  - P1-8 - TxD (an RS-232 output signal)

The Terminal Blocks have Pin-1 on the RIGHT

- TB1 Screw Terminal Block for RS-485 Interface
  - TB1-1 - Data- (B)
  - TB1-2 - GND
  - TB1-3 - Data+ (A)
- TB2 Screw Terminal Block for dual 4-20mA Sensor Inputs, Channels 2 and 3
  - TB2-1 - Power Out (+12V)
  - TB2-2 - 4-20mA Input Channel 2 (200 ohms to ground)
  - TB2-3 - 4-20mA Input Channel 3 (200 ohms to ground)
  - TB2-4 - GND
- TB3 Screw Terminal Block for 4-20mA Sensor Input Channel 1
  - TB3-1 - Power Out (+12V)
  - TB3-2 - 4-20mA Input Channel 1 (200 ohms to ground)
  - TB3-3 - GND
- 5-pin 0.1" header (**DNI**) is used for ICSP
  - J1-1 - Vpp / -MCLR
  - J1-2 - Vcc
  - J1-3 - GND

  - J1-4 - PGC
  - J1-5 - PGD
- LED DS1 indicates "Pump On"
- LED DS2 indicates "Communication"
- SW1 is an 8-position DIP Switch used for ModBus Slave Address Selection
  - SW1-1 - Address Bit 0, On = 1
  - SW1-2 - Address Bit 1, On = 2
  - SW1-3 - Address Bit 2, On = 4
  - SW1-4 - Address Bit 3, On = 8
  - SW1-5 - Address Bit 4, On = 16
  - SW1-6 - Address Bit 5, On = 32
  - SW1-7 - Unused, Must Be Off
  - SW1-8 - Unused, Must Be Off

TxAM Advanced Pump Control Features

With Operational Instructions

- Metal enclosure
- Works with HBT1, HBT2 Woodpecker TxAM Pumps.
- Has an expansion port to allow for field updates of software and for connection to main controller for external Modules to add variety of functions. These are small inexpensive modules connected by a plug-in cable allowing options such as Slave Pump to be controlled using this Controller and an external drive circuit.
- Expansion Interface to allow
  - ModBus Interface to a ModBus PLC
  - 4-20ma controller
  - external contact closure

The Controller may be configured for the following operation:

1. Cycles per minute
2. On Time /Off Time
3. Batch Timer (Quarts)
4. Quarts per Day
5. Gallons per day
6. Variable speed
7. ModBus Controlled (quarts/day, gallons/day, variable speed, On/OFF timer control) by optional Module
8. 4-20ma Controlled by optional Module

9. Remote contact control – on/off controlled

Five membrane switches are used for Pump operational control and a two line display is used to display information and prompts for pump set up. Additionally, the Controller may be operated solely by a 4-20ma loop, contact, or ModBus. While under ModBus control, only the Pause and test functions are operable. The Keypad Control Buttons (Membrane Switches) are as follows:

Start/Pause	Test/Prime	Up	Down	Enter

INITIAL SETUP

The initial setup is used to set the type of pump operation. This changes the Mode of Operation. To set an Initial Control Type you press and hold the **Start/Pause** Button while **`Txam Pumps, LLC`** is being displayed on the screen. (This is when the Controller is first powered up with the power switch.) Continue to hold down the **Start/Pause** Button and you will see the controller cycle through all the various Modes of operation. When you see the mode of operation you want, release the Button. The controller is now set for that mode of operation. Note: Releasing the Button will save the operational mode and the pump control will always power up to this mode unless it is changed again using the same procedure. This setting is usually set at the factory or during the initial pump setup.

SERVICE SETUP

This is done during initial pump installation and does not need to be repeated. After the **`Txam Pumps, LLC`** display times out (about 3 seconds) the display will prompt with a question **`Service Setup?`**

If you want to do an initial service setup press and release the **Up** Button. (You will have about five seconds to press the **Up** Button.)

If you do not press the **Up** Button the controller will begin operation.

Again, this is normally going to be a onetime set up. Once you enter the Service Setup, you will be prompted with questions. You will only be prompted with questions relating to the Type of Operation or Mode that has been set. When doing a Service Setup you will be using the **Up** and **Down** Buttons to select the correct answer.

When you have the correct answer press the **Enter** Button.

If you do not press the **Enter** Button or the **Up** or **Down** Button the pump control will wait about 30 seconds and then move to the next question.

If you want to select the currently displayed value press the **Enter** Button to move to the next question.

After all questions are answered you will see **`Service Complete`** displayed and then the Controller goes into the Operational mode. All these setup answers are saved.

Note: If you make a mistake during a set up, turn the power switch off and then back on then start over with the service setup.

If you do not press the **Enter** Button, the old setting will be kept even if you have moved up or down.

If you do not press a Button at anytime during set up, the display will time out and move on and normal operation will begin based on either the previous settings or any changes you have made.

CONTROL OPERATION

Normal Mode-Local Control

When the Controller is first powered up, you have the chance to make modifications to the Pump Control Type and Service Setup before operation begins. The Controller will display the Type of Pump Control and what the current settings are for operation. At this point the Controller will begin operation and display **`System ON`** .

Note the Service Set Up is not used for the setting of the run parameters, such as quarts per day or cycles per minute.

The Operating values are changed during the Normal Operation. When the pump is running you will see the display **`System ON`** .

When **`System ON`** is displayed the pump is being controlled by the Controller locally or remotely.

In order to make a change to the operating parameter press the **Start**/**Pause** Button. The Controller will now display **`Paused`** . While in the Pause mode you are able to change parameters.

Pressing the Start/Pause Button again will put you back to the **`System ON`** mode.

If you want to change the operating parameter put the Controller in Pause.

Now press the **Enter** Button and you will be prompted to Change a parameter. The **Up** and **Down** Buttons are used to change the parameter (i.e. Increase/Decrease quarts per day etc.). When you select the desired operating parameter, press the **Enter** Button.

The display will show OK change saved. **`OK change saved`** . The display will now show what the new operating parameter is set to.

Press the **Start**/**Pause** Button to resume operation and you will see **`System ON`** displayed.

Be sure you press the **Start** Button and see **`System ON`** .

If **`System ON`** is not displayed the Controller is not running but in a mode for making changes).

While **`System ON`** is displayed, press the **Pause** Button to make changes.

If you press the **Down** Button without pausing you will see **`Battery Voltage`** displayed.

The **Enter** Button has no function when **`System ON`** is displayed.

NOTE: If in a remote mode such as ModBus you cannot make changes to the setting but are able to Pause and Test/↩ Prime the Pump.

Test/**Prime**

Pressing the **TEST** Button will cause the pump to turn on for 30 seconds. After the 30 second time out the Controller resumes operation. (If the Controller was running it will continue to run, if the Controller was in Pause it will return to Pause).

During the test mode the display counts down the time remaining for the test operation.

You can press the **Pause** Button to abort the test mode.

This test mode is normally used to prime a pump or check for leaks. Testing will always run the pump at maximum speed.

Test mode will operate even in ModBus or 4-20ma control.

Connections The four spade lugs are required connections and the Spade lugs are above the display.

The lugs on the left are for Power (Battery) connections. The lugs are labeled positive and negative.

The Spade lugs on the right are for connection of the motor. (All Connection Lugs are marked on the Laminate)

ModBus, 4-20mA Control and Contact Closure require an external optional Module to operate

Note: Setup software is resident in the controller for optional external modules.

Advanced Controller Specific Mode Instructions

Cycles per minute –

1. Power on the unit and hold the **Start**/**Pause** Button until Cycles/Minute appears on the screen then release the **Start**/**Pause** Button.
2. The controller will display **`Service Setup?  Up=Yes`** . Push the **Up** Button and the screen will scroll and ask you some questions that relate to the Cycles/Minute mode. Answer the questions and press **Enter** to save. Then the Screen will display **`System ON`** .
3. Press the **Start**/**Pause** Button then press **Enter**

4. The screen will display Set Pump On Time and enter the number of seconds you want the pump to run and press **Enter**.
5. The screen will then display Set Cycles per Minute. Enter the number of cycles per you want the pump to run then press **Enter**. The screen will now display your settings.
6. Press the **Start**/**Pause** Button and the pump will operate as per your settings.

ON-OFF Timer –

1. Power on the unit and hold the **Start**/**Pause** Button until On Time/Off Time appears on the screen then release the **Start**/**Pause** Button.
2. The controller will display Service Setup? Up=Yes; Push the **Up** Button and the screen will scroll and ask you some questions that relate to the On Time/Off Time mode. Answer the questions and press **Enter** to save. The Screen will display **`System ON`** .
3. Press the **Pause** Button then press **Enter**
4. The Screen will display Set On Time (Between .5 Seconds and 120 Seconds) then press **Enter**
5. The Screen will display Set Off Time (Between .5 Seconds and 120 Seconds) then press **Enter**
6. The Screen will display your settings
7. Press the **Start** Button and the pump will operate as per your settings.

Batch Timer – Set how many Quarts are pumped during a single time and how many days a month it is done. Range is a batch of 1 to 100 quarts and done 1 to 28 time per month (per 4 week period)

1. Power on the unit and hold the **Start**/**Pause** Button until **`Batch Timer`** appears on the screen then release the Start/Pause Button.
2. The controller will display Service Setup? Up=Yes; Push the **Up** Button and the screen will scroll and ask you some questions that relate to the Batch Timer Mode. Answer the questions and press **Enter** to save. Then the Screen will display **`System ON`** .
3. Press the **Start**/**Pause** Button then press **Enter**
4. The Screen will display Set On Days (1-28 Days). Set the number of days in a month you want the pump to run then Press **Enter**.
5. The Screen will display Set Rate/Quarts (1-100 Quarts) then press **Enter**
6. The Screen will display your settings
7. Press the **Start**/**Pause** Button and the pump will operate as per your settings.

Quarts per Day – Sets how many quarts are pumped per day. The calculations decide the ON time and OFF times required each minute in order to pump the selected Quarts per day. Range is 1 to 24 Quarts

1. Power on the unit and hold the Start/Pause Button until Quarts per Day appears on the screen then release the Start/Pause Button.
2. The controller will display Service Setup? Up=Yes; Push the **Up** Button and the screen will scroll and ask you some questions that relate to the Quarts per Day Mode. Answer the questions and press **Enter** to save. The Screen will display **`System ON`** .
3. Press the **Start**/**Pause** Button then press **Enter**

4. The Screen will display Set Quarts/Day
5. Press the **Up** or **Down** Button to set the desired quarts per day then Press **Enter**
6. The Screen will display your settings
7. Press the **Start**/**Pause** Button and the pump will operate as per your settings.

Gallons per Day – Sets how many gallons are pumped per day. The calculations decide the ON time and OFF times required each minute in order to pump the selected Gallons per day. Range is 1 to 24 Gallons.

1. Power on the unit and hold the **Start**/**Pause** Button until Gallons per Day appears on the screen then release the **Start**/**Pause** Button.
2. The controller will display Service Setup? Up=Yes; Push the **Up** Button and the screen will scroll and ask you some questions that relate to the Gallons per Day Mode. Answer the questions and press **Enter** to save. Then the Screen will display **`System ON`** .
3. Press the **Start**/**Pause** Button then press **Enter**
4. The Screen will display Set Gallons/Day (Range 1-24 Gallons)
5. Press the **Up** or **Down** Button to set the desired Gallons per Day then Press **Enter**
6. The Screen will display your settings
7. Press the **Start** Button and the pump will operate as per your settings.

Variable Speed – Set the speed of the motor. Range is 1 to 10 (with 10 being full speed)

1. Power on the unit and hold the **Start**/**Pause** Button until **`Variable Speed`** . appears on the screen then release the **Start**/**Pause** Button.
2. The controller will display **`Service Setup? Up=Yes`** Push the **Up** Button and the screen will scroll and ask you some questions that relate to the Variable Speed Mode. Answer the questions and press **Enter** to save. Then the Screen will display **`System ON`** .
3. Press the **Start**/**Pause** Button then press **Enter**
4. The Screen will display **`Set Pump Speed (Range 1-10)`** .
5. Press the **Up** or **Down** Button to set the desired Pump Speed then press **Enter**
6. The Screen will display your settings
7. Press the **Start** Button and the pump will operate as per your settings.

The pump will always turn off when the battery voltage gets low. This will be indicated on the display along with the current battery voltage. When the battery charges it will automatically start back running.

Additional Notes

In BATCH MODE if you pause or test the system, when you turn the system back on it will resume with the off time. This is to prevent the pump from running another batch cycle. If you need it to run another batch just cycle the power to the Pump Control.

In VARIABLE SPEED mode anytime the motor is stopped it will always start at full speed for two seconds to overcome the torque of the system and then change to the variable speed setting.

All settings are saved once they are set.

Defaults have been set for operation.

Questions asked during Service Setup are based on the operation and may ask such things as motor type, stroke length, plunger size, dual or single head pump and pressure range. These parameters are used to calculate gallons and quarts. Also the Controller may ask to Enable/Disable Temperature control in which case you will be prompted for the ON and OFF temperatures. You may also be asked for Battery Saver mode Enable/Disable. This is used to slow down pump operation as the battery discharges to allow for longer pump operation.

ModBus Communication and Control

Register info for Commands and Data

RTU Slave

Read Discrete Inputs Function code 02 – state of operation. low battery, local control, temp inhibit, etc.

Read Holding register Function code 03 – read Timer Type and setting

Read Input register Function code 04 – read information and data

Write Single Coil Function code 05 – turn On/Off pump

Write Single Holding Register Function code 06 – Writes Timer Type and operational parameters

Control Pump ON/OFF via the Write Single coil Register(05)

This Controls Advanced Pump Control operation

coil 1 is used other coils not used, Coil 1 ON => Pump Run OFF => Pump off (coil 1 = address 0)

Set Pump Operation using Write Single Holding Register (06)

This Controls Advanced Pump Control operation

Address 0 sets the Timer Type (mode of operation), modes of operation settable via ModBus command

Mode	Value	Description
00	CpM	cycles per minute operation (00 is the value to write)
01	On/OFF	on-off timer operation
02	QpD	quarts per day operation
03	GpD	gallons per day operation
04	VS	variable speed operation
05	BT	Batch timer mode of operation-quarts/day and days on

Address 1 sets parameter associated with the mode of operation (still function 06)

number of cycles per minute -time to run in seconds and duration-time the pump runs (2 numbers cctt) (number such as hex 0205 for 2 cycles per minute and 5 seconds duration)

Off time in 1/2 seconds -On time in 1/2 seconds (2 numbers required?4 digits) Note that the time you enter is divided by 2 to allow half seconds settings to be made

Number of quarts per day (number such as 7 for 7 quarts per day)

Number of gallons per day (1 number required)

Days ON in 28 day batch cycle and batch quarts each cycle (2 numbers required)

motor speed 1 to 10

BELOW COMMANDS ARE INFORMATION AND NOT CONTROL Read values and operational parameters using Read Input register (04) input_regs[0] Battery voltage (see Note 1) input_regs[1] temperature probe (see Note 2) input_regs[2] temperature set points TEMPHI TEMPLO input_regs[3] External 4-20ma signal channel #1 (see Note 3) input_regs[4] Operational parameters∗ defined below input_regs[5] External 4-20ma signal channel #2 input_regs[6] External 4-20ma signal channel #3 input_regs[7] SW version (internal housekeeping)

∗∗Parameters indicated in input_regs[4] (binary decode of the 16 bits required) Bit 0 – reserved Bit 1–plunger size 1/4 Bit 2–plunger size 3/8 Bit 3–plunger size 1/2 Bit 4–stroke length full Bit 5–stroke length three quarter Bit 6–stroke length one half Bit 7–dual head pump or single head pump (0 = single, 1 = dual) Bit 8–voltage cutback active = 1 Bit 9–temperature shutdown active = 1 Bit 10–motor 1 HBT 1 33 RPM Bit 11–motor 2 HBT 2 66 RPM Bit 12–motor 3 Wood Pecker 43 RPM Bit 13–motor 4 Other (ie AC motor using AC module) Bit 14–reserved Bit15?reserved

Read the alarm status/state of operation using the Read Discrete Input (02) Addr0– Temperature sensor fault or No temperature sensor connected (1 = fault or no sensor) Addr1–System is requested to Run (1 = On) Addr2–Initialization complete –communication between AdvPC and Module established Addr3–Battery voltage OK – above 10.8 volts Addr4–Temperature inhibiting operation Addr5–Voltage cutback operation-in power saving mode Addr6–Blown fuse detected–no power to Motor Addr7?Bad mode – set if command sent was an invalid Timer Type (Func 06)-pump will stop

Read Holding register Function (03) – read timer type and setting holding_regs[0] Timer Type currently running – this is value you wrote to Addr 0 (funct 06) holding_regs[1] operating parameters 1 and 2– this is value you wrote to Addr 1 (funct 06)

NOTES: Most commonly used will be function 5 and 6 to control timer–others provide information. When you power up, the Controller will start running using the last values written via ModBus Anytime you write a new Timer Type the pump will turn off, after you write the operating parameters you must restart the Controller. (use write to Coil 05) You can change the settings (parameters) while the pump is running. (example gallons per day) If you write an operating parameter that is too large it will be set to largest allowable operating value

When using Write Single Register(ModBus function code 6) additional information about data Address 0 is the Timer Type, ie CpM or quarts/day Address 1 parameter 1 and 2∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗MSB/LSB MSBYTE /LSBYTE of 16 bit word num_cycles / time2run number of cycles per minute, time to run in seconds-duration ------— / quarts number of quarts per day ----— / gallons number of gallons per day timeoff / timeon off times in 1/2 seconds on time in 1/2 seconds dayson / bgal Days on in 28 day batch cycle, batch quarts each cycle ------— / speed motor speed 1 to 10

When using ModBus poll (see below) Read/Write definitions set quantity to 8 (default is 10) also scan rate 1000 and display in HEX

Baud Rate 9600, 1 stop bit, no parity, ModBus type – RTU Slave Dip switch sets ModBus address Dip Switch 1- 6 is binary decode of address (Examples below) DIP Switch Position 6 5 4 3 2 1 ModBus Address off off off off off off 0 off off off off off On 1 off Off off off On off 2 off off off off On On 3 off off off On off off 4 off off off On off On 5 off off off On On off 6

Dip Switch position 7 and 8 reserved (Dip Switch 8 MUST BE IN THE OFF POSITION)

NOTE 1: BATTERY VOLTAGE

To get battery voltage you read an number from 0 to 255. To covert this to battery voltage multiply the number by 0.06

(reading X 0.06) = battery voltage example (192 X .06) = 11.52 volts, and 213 = 12.78 volts

Note 2: TEMPERATURE

To get temperature is a little more complicated To get temperature you read a number from 41 to 1024, this is equal to -45C to +75C First 0(zero) degrees C is a reading of 409 then each 1 degree C is 8 counts

for example 409 = 0 degree C 417 = 1 degree C 425 = 2 degree C 401 = -1 degree C 393 = -2 degree C

Note 3: 4 TO 20 MA LOOPS

To convert the 4 to 20 milliamp reading It is linear relation with 0 = 0 ma and 1024 = 25 ma therefore each ma = 41 counts so 4 ma = 164 , 12ms = 492 and 20 ma = 820

Note 4: Example of writing to a coil

Decode the follow data stream-> 01 05 00 01 FF 00 DD FA

01: The Slave Address (01 = 01 hex) 05: The Function Code (Force Single Coil) 0001: The Data Address of the coil. (coil# 2 - 1 = 1 = 01 hex) FF00: The status to write (FF00 = ON, 0000 = OFF) DDFA: The CRC (cyclic redundancy check) for error checking. We use coil # 1 for ON/OFF control – the above example is writing to coil#2 Coils are addressed starting at zero–coil 1 is addressed as 0

ModBus Module Connections

Writing ModBus Parameters

The ModBus Module Program will limit the setting for each Timer Type. This is to prevent ModBus from entering invalid parameters for a mode. If you attempt to write a number larger or smaller than the set point, the program will limit the number to be in the correct range. Allowable operational ranges are as follows:

- Quarts per Day range: 1 to 24 quarts (0x01 to 0x18 Hex)
- Gallons per Day range: 1 to 24 gallons (0x01 to 0x18 Hex)
- Batch Timer range: 1 to 100 quarts (0x01 to 0x64 hex)
- Cycles per Minute range: Cycles 1 to 10 cycles per minute (0x01 to 0x0a hex) ON time 1 to 5 seconds
- Variable speed range: 0 to 10 (0x01 to 0x0a hex) 0 = OFF, 10 = full speed
- On Time/Off Time Range: (on/ off times in half seconds) On Time 1 to 180 half seconds (0x01 to 0xb4 hex) (equals 90 seconds max) Off Time 1 to 240 half seconds (0x01 to 0xF0 hex) (equals 120 seconds max)

To check what you wrote: Turn the controller off and back on. It will display ModBus timer and show the settings as it powers up.

Note: The Green LED flashes each second as a heartbeat to indicate the communication module is operational. The Red LED is on if ModBus has requested the Controller to run and will be off if the ModBus has requested the Controller to stop operation. Even though the Red LED is on the pump may not be running, the LED is just indicating the Controller is in the run mode (it could be in an off time cycle). If the Red LED flashes each few seconds, this indicates the module is trying to establish communication with the Advanced Pump Control. Red LED flashes fast (each 1/4 second) if a communications failure is detected.

Note: It is possible in quarts per day and gallons per day mode to write a value small enough that the pump does not run, this is based on the plunger, stroke and motor. If the calculation indicates that the pump will run less than one second per minute you will not see the pump run.

Additional Information on changing the type of pump operation:

To Set Pump Operation Mode use Write Single Holding Register (06). This controls the AdvPC operation.

Writing to Address 0 sets the Timer Type (mode of operation), the modes of operation settable via ModBus command are:

Mode	Value	Description
00	CpM	cycles per minute operation (00 is the value to write)

01	On/OFF	on-off timer operation
02	QpD	quarts per day operation
03	GpD	gallons per day operation
04	VS	variable speed operation
05	BT	Batch timer mode of operation-quarts/day and days on

To change the mode of operation, write a number from 0 to 5 according to the table above. When you write a new mode this stops the pump from running. After you set the new parameters via address 1 you must restart the pump by writing a coil command. If you are only writing new parameters (address 1) and do not write (change) the Timer Type (address 0) the pump does not stop and there will be no need to restart the pump. If you write an invalid Timer Type (number larger than 5) the pump will turn off and a valid type must be written before you can restart it.

When using Write Single Register (ModBus function code 6) Address 0 Additional information/ data is required.

Once you change the Timer Type you must also write a new set of operational parameters. Also use this procedure for just changing parameters.

Write Single Holding Register (06) Address 1 to set a new parameter.

Address 1 sets parameter associated with the mode of operation (still function 06)

The numbers you write to address 1 are decoded based on the Timer Type.

Address 0 is written to set Timer Type, ie CpM or quarts/day

Address 1 writes parameter 1 and 2 for the selected Timer Type. Since the register is 16 bits it has a MSB/LSB and is much easier to think of it a two eight bit numbers UpperBYTE /LowerBYTE of 16 bit word (hex) ------— / quarts number of quarts per day ------— / gallons number of gallons per day ------— / speed motor speed 1 to 10 num_cycles / time2run number of cycles per minute, time to run in seconds-duration timeoff / timeon off times in 1/2 seconds on time in 1/2 seconds dayson / bgal Days on in 28 day batch cycle, batch quarts each cycle

The simplest parameters to write are when using quarts per day, gallons per day and variable speed. You need only to write a simple number from 1 to 24 for quarts or gallons or 0 to 10 for variable speed. Example: while in quarts per day if you were to Write Single Holding Register (06) Address 1 the number 7 you will be operating at 7 quarts per day, if you write the number 17 (hex 0x0011) you would be operating at 17 quarts per day. If you write a number larger than 24 (0x0018 hex) it will default to the maximum of 24 quarts per day.

As an example for the cycles per minute timer, think of the hex word (16 bits) as two bytes [cycles per minute] and [duration in seconds] as a 16-bit hex number 0xCCTT.

If you wanted 6 cycles per minute and 5 seconds duration you would write the hex number 0x0605 (decimal 1541).

If you wanted 7 cycles and 5 seconds write hex 0x0705 (decimal 1797)

If you wanted 10 cycles/3 seconds you would write hex 0x0a03 (decimal 2563).

If you wanted 3 cycles/1 seconds you would write hex 0x0301 (decimal 769).

If you wanted 4 cycles/4 seconds you would write hex 0x0404 (decimal 1028).

You can see from the above you need to think of the two parameters as a hex number.

Chapter 6

Ullage Sensor User Guide

Command Overview

Command Sentences are received from the serial port. They may originate with a user, installer, computer data collection link or manufacturing test station.

The same command structure is used by stored commands within the Sensor. These commands are automatically executed when triggered by an event such as a timer, contact closure, or control panel request.

Command Sentences consist of one or more individual commands (with their optional parameters) followed by a checksum. All command and response characters are printable ASCII and designed to be human readable and (somewhat) intuitive. No pure binary data communication is expected or allowed. Command sentences may be up to 80 characters in length. It is generally expected that command sentences will be separated by line separators such as ASCII CR, LF, etc. Sending a CR before every command may be required in some situations, such as half-duplex RS-485, to ensure that the receive buffer properly frames the sentence and does not fail the checksum unnecessarily.

Individual commands consist of a single prefix character with optional numeric parameters. Numeric values may be either decimal or hexadecimal. Multiple numeric parameters are separated by commas and terminated by a semi-colon.

Commands may cause the reporting of a particular value, the setting of an internal value or the testing of a particular condition.

Reporting. Various values that are measured, calculated or stored in the Sensor may be reported. This example reports the measured values of the two pressure transducers (ambient and ullage) and the computed the difference between the two. Note that all response values are reported in decimal engineering units with appropriate signs and decimal point.

Example Command	Example Result
`a?;b?;d?`	`a:1015.5mB;b:1015.5mB;d:+0.0mB`

Setting. There are several internal operation modes that can be set remotely, as well as sensor calibrations and the real-time clock. The "=" before the parameter indicates that a new value is to be stored. The command may have multiple parameters separated by commas to allow for more convenient and intuitive entry.

Example Command	Example Result
`T=12,34,56;T?;d?`	`T:13-06-13,12:34:56;d:+0.0mB`

Testing. Sensors may be used on multi-drop communication links (such as RS-485) or with broadcast protocols. Conditional commands allow Sensor units to be individually selected. Conditional commands are placed at the beginning of a command Sentence. If any conditional fails, the entire command Sentence is discarded (ignored).

Example Command	Example Result
S∼156;s?;d?	S:156,0;d:+0.0mB

This would come from the Sensor with serial number 156 and no other unit would respond.

In general, the "∼" character is used to indicate a conditional. The parameter following the "∼" is compared to the value stored internally and if the values do not match the test fails and the remainder of the command Sentence is discarded.

There are two special cases for conditional commands. The first is the checksum which is placed at the end of the command Sentence. If the checksum is present and does not match the value computed internally, the entire command Sentence is discarded. Checksums are optional if commands are being echoed, since we assume that echoing is being used by a person manually entering commands.

The second special case is command Sentences containing Sequence numbers. The "s" command specifies a command sequence number which must NOT match the previous sequence number. This is intended for use with unreliable communication links which may duplicate packets or drop responses, causing the host to send retries. This ensures that command Sentences that should be executed only once (such as incrementing the hour for Daylight Saving Time) can be sent over unreliable data links and yield the expected results.

Notes About Checksums

To help ensure valid communication checksums are used at the end of every Command or Response Sentence. These checksums follow the general format used by the National Marine Electronics Association and should be familiar to anyone who has worked with a GPS unit.

The checksum is transmitted as an ASCII "∗" character followed by two ASCII hex characters representing the hexadecimal value of the eight-bit checksum.

All characters from the beginning of the Sentence up to and including the "∗" are summed.

Non-Volatile Operating Parameters

The table shows the parameters that may be set to configure the Sensor for particular applications, or that may be used in response Sentences. These parameters are stored in the Sensor and are preserved in the event of power loss.

Parameter	Description	Examples
eeAlias	Customer's Sensor Serial Number Alias	
eeTimer	Automatic reporting interval in 1/8 seconds	
eeRxSeq	Command Sentence sequence number	
eeTxSeq	Response Sentence sequence number	
eeShow	Automatic Reporting Command Sentence	
eeFormatTime	Bits controlling the way the Date and Time are reported	
eeFlags	Bits controlling the displayed data formats	

The following non-volatile items are used internally and are not generally available to the end user. This table is included only to provide a complete picture of the Sensor internal operation. These parameters are stored permanently during the manufacturing process.

Parameter	Description	Examples
eeSerial	Sensor Serial Number 156	
eeVersion	Sensor Firmware Version Number	1.1.44
eeDate	Sensor Manufacturing Date	11Jun2013
eeID	Sensor Model Number and description	Ullage Sensor
eeVcc	Core Internal Regulator Calibration	4970
eeBattScale	Scale factor for external supply voltage	604
eePresScale	Scale factor to convert differential pressures	40
eePresOffset	Offset to compute absolute pressures	9550
eeCount	Reset Count	5

Individual operating modes are controlled by bits in the eeFormatTime and eeFlags values. These operating parameters have a command that allows them to be set individually or collectively, or viewed as a whole. Some of the bits are dependent on other parameters to complete their meaning: for example, Setting a particular date-display Order will do nothing unless you also request that the date actually be displayed.

The internal names and meanings of individual bits are described in the next tables.

Time Format Bit	Description	Command	Meaning
fFormatTime	Include time in result. If the clock has not been set since power-on, this will be elapsed time.		
fFormatDate	Include date in result. If the date has not been set since power-on this will be an integer number of elapsed days.		
fFormatDelim	Include Delimiters `"-"`, `","` and `":"` if needed		
fFormatMDY	Display Month, Day, Year order		
fFormatDMY	Display Day, Month Year order		
fFormatDD	Display days only (suppress month and year).		
fFormatMMM	Display the month name as three characters		
fFormatMS	Include milli-seconds in the displayed time value		

This table contains several examples of Time Format Byte values and the results that can be expected in the output. The Time Format byte that we are describing here sets the default format used by the Response Sentences and the user interface display. These same byte values may also be used in the "T" Command to override the default behavior and get the desired format.

Time Format Byte	Command	Resulting Format
52	`z=FF005200`	`25Dec13`
02	`z=FF000200`	`121225`

0E	z=FF000E00	12-25-13
03	z=FF000300	121225201918
07	z=FF000700	13-12-25,20:19:18
01	z=FF000100	201918
05	z=FF000500	20:19:18
83	z=FF008300	131225201918.500
87	z=FF008700	13-12-25,20:19:18.500

The operating Mode Flag bits are defined in the following table.

Mode Flag Bit	Description	Command	Meaning
feeMetric	Report results using metric units	z=00030000 z=00030001 z=00030002 z=00030003	PSI mBar . .
feeAltUnit	Use alternate units of measure		
fee	(currently unused)		
feeUnits	Display units of measure after every value	z=00100000	Units Off
		z=00100010	Units On
feeVolts	Display analog voltage associated with every reading	z=00100000	Volts Off
		z=00100010	Volts On
feeCounts	Display raw analog value associated with every reading	z=00200000	Raw Off
		z=00200020	Raw On
feeNVpair	Include the parameter name character and an = at the beginning of every value reported	z=00400000 z=00400040	Name Off Name= On
feeEcho	Echo characters received from the serial port. This is intended for use with manually entered commands from a terminal program. Setting this also suppresses the checksum on the end of response Sentences.	z=00800000 z=00800080	Echo Off Echo On

Serial Port and Command Reference

Cmd	Parameters	Description	Examples
a		Display pressure channel A	
b		Display pressure channel B	

c		Display pressure channel C (optional)	
d		Display differential pressure A-B	
D	`D=y,m,d`	Set the date	`D=13,5,30`
D		Display the date like "23;" or "13-05-30;"	D
f	`f`	Display the presence of a fault condition	
g	`gnn,nn,nn...`	Display one or more memory values	
		RAM bytes in the range 0..1FF	
		EEPROM bytes in the range 200..2FF	
i	`i=sss,f`	Set the reporting interval in seconds and 1/8ths	
I		Send the bootstrap ID string.	I
m		Send the programmed Response Sentence	m
p	`p=hh`	Set the pump measurement cycle phase. Generally used to start a measurement.	p=1
p		Display the pump measurement cycle phase.	p
r		Manually force a Sensor reset	r
t		Test command used for various things during development	t
T	`T=hh,mm,ss`	Set the clock. The individual parameters are optional, and are not range-checked or checked for internal consistency	T=12,34,56

Sensor Control Panel

The Sensor may optionally be equipped with a user-interface control panel. The panel includes a LCD display and buttons that allow an operator to control the sensor and observe its readings. The interface is a simple menu tree controlled by UP, DOWN and SELECT buttons.

Some Display Items allow numeric values to be entered. Pressing SELECT will cause the value to blink. Use the UP and DOWN buttons to change the value, then press SELECT again to save the new value.

Item	Menu Display	Notes
0	Ullage Test BKMcM.com(c)2013	
1	Ullage Test Day 0 10:23:45	
2	Ullage Test Speed 45	
3	Serial	
4	Ullage Test Current 75	
5	Diag NoFault	
6		
7		
8		

Chapter 7

Module Index

7.1 Modules

Here is a list of all modules:

Chapter 8

Data Structure Index

8.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 9

File Index

9.1 File List

Here is a list of all files with brief descriptions:

Chapter 10

Module Documentation

10.1 Control-Data Module Controller

CDMcontroller Description of the Control-Data Module

Modules

- Analog to Digital Conversion and result output

  Handle selection and polling of multiple A/D Converter channels.
- CDMA Cellular Radio Modem Protocol

  Handle SMS Messages using CDMA Hardware.
- Module Command and Response Protocol

  Generate and parse command sentences.
- Non-volatile memory allocation

  Memory allocation for non-volatile memory.
- Human-readable Input and Output Formatting

  Input and output conversion for ring buffers.
- GSM Cellular Radio Modem Protocol

  Handle SMS Messages using GSM Hardware.
- The Interrupt Service Routines
- User Interface Menu

  Menu for the LCD Display Interface.
- Support Functions

  Various support utilities.
- ModBus Functions

  ModBus Support includes Master and Slave in both ASCII and RTU modes.
- Power-Clock-Data Bus Operation

  The PCD-Bus provides for power supply and message exchange between modules.
- Peripheral Device Polling

  Polling of hardware peripherals that are not handled by interrupts.
- Power Management

  Handling of peripheral power and Sleep Mode.
- Ring Buffers

Ring Buffers allow consistent access to strings of characters in different types of memory or devices or messages.

- Real-Time Clock

  Maintenance and display of the Real-Time Clock and Timers.
- Module Hardware Styles

  Handle PCD-Bus Modules with different hardware styles.
- UART Communication

  Handle Serial I/O to a variety of peripheral devices.
- User Interface

  Provides user interaction via a LCD Display and buttons or scroll wheel.

Data Structures

- struct FlagBits_t

  FlagBits_t *Internal Status and Control Bits*
- struct CompareBits_t

  CompareBits_t *Ring-based String Comparison Status and Control Bits*
- struct ErrorBits_t

  ErrorBits_t *Ring Buffer Error indication bits*

Macros

- #define versionNumber (0x01010105)

  Firmware Version Number.
- #define versionDate (0x20141019)

  Firmware Compilation Date.
- #define leftLED (LATAbits.LA4)

  Output bit that controls the Left LED.
- #define leftLEDtris (TRISAbits.RA4)

  TRIS bit that controls the Left LED.
- #define rightLED (LATAbits.LA5)

  Output bit that controls the Right LED.
- #define rightLEDtris (TRISAbits.RA5)

  TRIS bit that controls the Right LED.
- #define RTSout (LATCbits.LC0)

  Output bit for Modem RTS and Turns on RS-485 Drivers.
- #define RTSouttris (TRISCbits.TRISC0)

  TRIS bit should be 0 for output.
- #define CTSin (PORTCbits.RC1)

  Input bit for Modem CTS.
- #define CTSintris (TRISCbits.TRISC1)

  TRIS bit should be 1 for input.
- #define DSRin (PORTCbits.RC2)

  Input bit for Modem DSR.
- #define DSRintris (TRISCbits.TRISC2)

  TRIS bit should be 1 for input.

- ModeBits_t ModeBits

  Non-Volatile shadow at nvMode (16 bits)

10.1.1 Detailed Description

CDMcontroller Description of the Control-Data Module

The Universal Intelligent Sensor Controller provides a standardized method of sensing, reporting and controlling a variety of devices.

10.1.2 Macro Definition Documentation

10.1.2.1 #define CTSin (PORTCbits.RC1)

Input bit for Modem CTS.

Definition at line 27 of file cdm.h.

10.1.2.2 #define CTSintris (TRISCbits.TRISC1)

TRIS bit should be 1 for input.

Definition at line 28 of file cdm.h.

10.1.2.3 #define DSRin (PORTCbits.RC2)

Input bit for Modem DSR.

Definition at line 30 of file cdm.h.

10.1.2.4 #define DSRintris (TRISCbits.TRISC2)

TRIS bit should be 1 for input.

Definition at line 31 of file cdm.h.

10.1.2.5 #define leftLED (LATAbits.LA4)

Output bit that controls the Left LED.

Definition at line 19 of file cdm.h.

10.1.2.6 #define leftLEDtris (TRISAbits.RA4)

TRIS bit that controls the Left LED.

Definition at line 20 of file cdm.h.

10.1.2.7 #define rightLED (LATAbits.LA5)

Output bit that controls the Right LED.

Definition at line 21 of file cdm.h.

10.1.2.8 #define rightLEDtris (TRISAbits.RA5)

TRIS bit that controls the Right LED.

Definition at line 22 of file cdm.h.

10.1.2.9 #define RTSout (LATCbits.LC0)

Output bit for Modem RTS and Turns on RS-485 Drivers.

Definition at line 24 of file cdm.h.

10.1.2.10 #define RTSouttris (TRISCbits.TRISC0)

TRIS bit should be 0 for output.

Definition at line 25 of file cdm.h.

10.1.2.11 #define versionDate (0x20141019)

Firmware Compilation Date.

Definition at line 17 of file cdm.h.

10.1.2.12 #define versionNumber (0x01010105)

Firmware Version Number.

Definition at line 16 of file cdm.h.

10.1.3 Variable Documentation

10.1.3.1 ModeBits_t ModeBits

Non-Volatile shadow at nvMode (16 bits)

Definition at line 347 of file cdm.h.

10.2 Analog to Digital Conversion and result output

Handle selection and polling of multiple A/D Converter channels.

Functions

- void adcPoll (void)

  adcPoll *Poll the enabled A/D converter channels and convert readings*
- void adcDisable (void)

  adcDisable *Disable all ADC operation for Sleep Mode*
- void adcShowChannel (uint8_t channel)

  adcShowChannel *Show the formatted value of the selected Analog channel*
- void adcShow (uint8_t channel)

  adcShow *Show the current value of the selected ADC channel or all enabled channels*

10.2.1 Detailed Description

Handle selection and polling of multiple A/D Converter channels.

10.2.2 Function Documentation

10.2.2.1 void adcDisable (void)

adcDisable Disable all ADC operation for Sleep Mode

Definition at line 1100 of file cdm.c.

```
1101 {   // **adcDisable** Turn off ADC and Vref for sleep mode
1102     ADCON0 = 0x00;
1103     VREFCON0 = 0x00;
1104 }
```

10.2.2.2 void adcPoll (void)

adcPoll Poll the enabled A/D converter channels and convert readings

Definition at line 1082 of file cdm.c.

```
1083 {   // **adcPoll** Poll the enabled A/D converter channels and convert readings
1084     uint8_t adcChannel = (ADCON0 >> 2) & 0x07;  // Previous Channel
1085     if (ADCON0bits.ADON) {  // Read the result from the current channel
1086         if (ADCON0bits.GO) return;  // Conversion in progress
1087         adcResult[adcChannel] = ADRESH * 256 + ADRESL;
1088     } else {    // Initialize the Voltage Reference and ADC module
1089         VREFCON0 = 0xF0;    // Enable the 4.096v internal reference
1090         ADCON1 = 0x08;  // Use internal Fixed Voltage Reference
1091         ADCON2 = 0xBF;  // Right Justified Results and very slow conversion clock
1092         ADCON0bits.ADON = 1;    // Turn on the ADC
1093     }
1094     adcChannel = (adcChannel + 1) & 0x07;   // Step to the next channel
1095     ADCON0 = (adcChannel << 2) + 1;
1096     // If the channel is analog we start the Acquisition and conversion
1097     if (ANSELA & (1 << adcChannel)) ADCON0bits.GO = 1;
1098 }
```

10.2.2.3 void adcShow (uint8_t *channel*)

adcShow Show the current value of the selected ADC channel or all enabled channels

Parameters

channel	Channel number to display or 0xFF for all

Definition at line 1118 of file cdm.c.

```
{   // **adcShow** Show the current value of the selected ADC channel or all enabled channels
    if (channel == 0xFF) {  // Display all active channels in order
        for (channel=0; channel <= 7; channel++) adcShowChannel(channel);
    } else adcShowChannel(channel);
}
```

10.2.2.4 void adcShowChannel (uint8_t *channel*)

adcShowChannel Show the formatted value of the selected Analog channel

Parameters

channel	

Definition at line 1106 of file cdm.c.

```
{   // **adcShowChannel** Show the formatted value of the selected Analog channel
    uint16_t mV;
    if (ANSELA & (1 << channel)) {
        mV = 20 * adcResult[channel];
        putInt(mV / 1000, 0);
        put('.');
        putInt(mV / 10, 2);
        if (ModeBits.fUnits) put('v');
    }
}
```

10.3 Module Command and Response Protocol

Generate and parse command sentences.

Functions

- uint8_t isAlpha (uint16_t ch)

  isAlpha *If the character ch is in the SixBit alphanumeric set return true*
- void showVersion (uint8_t n)

  showVersion *Show Version and Version Date as selected by parameter bits*
- void getPhrase (ringSelect_t ringT, ringSelect_t ringF)

  getPhrase *Copy the next delimited command phrase from ringFrom to the scrRing*
- void putFieldDelimiter (void)

  putFieldDelimiter *Put the field delimiter in the result ring with possible leading sender ID*
- void putSimpleFieldID (uint8_t ch)

  putSimpleFieldID *Output a properly formatted Field Identifier (like ',S:')*
- void putIndexedFieldID (uint8_t ch, uint16_t n)

  putIndexedFieldID *Output a properly formatted Field Identifier (like ',a3:')*
- uint8_t doPhrase (ringSelect_t ringT, ringSelect_t ringF)

  doPhrase *Process the single Command that is already isolated in ringF*
- void doSentence (ringSelect_t ringT, ringSelect_t ringF)

  doSentence *Verify the checksum and process the sequence of phrases*

Variables

- uint8_t lastCommandSeq

- ModeBits_t ModeBits

  Non-Volatile shadow at nvMode (16 bits)
- SerialBits_t SerialBits

  Non-Volatile shadow at nvSerial (16 bits)
- ModbusBits_t ModbusBits

  Non-Volatile shadow at nvModbus (8 bits)
- FlagBits_t FlagBits

  Operating Status Flag Bits.
- CompareBits_t CompareBits

  Control and results of string comparisons.
- ErrorBits_t ErrorBits

  Global Error indication flags.
- #define delimCS ('∗')

  Delimiter for Checksum.
- #define delimPhrase (';')

  Delimiter for Command Phrases.
- #define delimSubParam (',')

  Delimiter for Sub-Parameters.
- #define delimAssign ('=')

Delimiter for Parameter Value Assignment.

- #define delimQuery ('?')

  Delimiter for Parameter Value Query.
- #define delimReport (':')

  Delimiter for Parameter Value Report.
- #define delimTest ('∼')

  Delimiter for Parameter Value Test.
- #define delimQuot1 (0x22)

  Delimiter for Quoted Strings - Double Quote '"'.
- #define delimQuot2 (0x27)

  Delimiter for Quoted Strings - Single Quote "'".
- #define delimQuot3 (0x60)

  Delimiter for Quoted Strings - Accent Grave '`'.
- #define cmdSerial ('S')

  Command Serial Number.
- #define cmdSequence ('s')

  Command Sequence Number.
- #define cmdVersion ('v')

  Command Version Number.
- #define cmdHardware ('H')

  Command Hardware Model String.
- #define cmdStyleCode ('h')

  Command Hardware Style Code in Hex.
- #define cmdID ('I')

  Command Identification String.
- #define cmdInterval ('i')

  Command Interval Timer.
- #define cmdForward ('F')

  Command Forward text to PCD-bus.
- #define cmdReport ('r')

  Command Report Format.
- #define cmdDest ('R')

  Command Report Destination (like a phone number)
- #define cmdAnalog ('a')

  Command Analog Voltage.
- #define cmdMode ('m')

  Command Mode Bits.
- #define cmdMemory ('M')

  Command Memory Dump.
- #define cmdDate ('d')

  Command Date.
- #define cmdTime ('t')

  Command Time.
- #define cmdOperation ('o')

  Command Initiates the canned operations.
- #define cmdDiag ('z')

  Command Various Diagnostics.

- #define cmdPlaceholder ('%')

  Command (placeholder for future commands)
- #define SixtyTwo (0x24)

  Sixbit character 62 is ASCII '$'.
- #define SixtyThree (0x5F)

  Sixbit character 63 is ASCII '_'.
- enum uiState_t {
  none, up, down, enter,
  back, holding }

  ***uiState_t** Allowed States for Buttons or Scroll Wheel*
- enum cdState_t {
  cdWaitIdle, cdIdle, cdRecvData, cdTrmtData,
  cdRecvDone, cdTrmtDone }

  ***cdState_t** Allowed States for the PCD-Bus Interface*

10.3.1 Detailed Description

Generate and parse command sentences.

10.3.2 Macro Definition Documentation

10.3.2.1 #define cmdAnalog ('a')

Command Analog Voltage.

Definition at line 252 of file cdm.h.

10.3.2.2 #define cmdDate ('d')

Command Date.

Definition at line 255 of file cdm.h.

10.3.2.3 #define cmdDest ('R')

Command Report Destination (like a phone number)

Definition at line 251 of file cdm.h.

10.3.2.4 #define cmdDiag ('z')

Command Various Diagnostics.

Definition at line 258 of file cdm.h.

10.3.2.5 #define cmdForward ('F')

Command Forward text to PCD-bus.

Definition at line 249 of file cdm.h.

10.3.2.6 #define cmdHardware ('H')

Command Hardware Model String.

Definition at line 245 of file cdm.h.

10.3.2.7 #define cmdID ('I')

Command Identification String.

Definition at line 247 of file cdm.h.

10.3.2.8 #define cmdInterval ('i')

Command Interval Timer.

Definition at line 248 of file cdm.h.

10.3.2.9 #define cmdMemory ('M')

Command Memory Dump.

Definition at line 254 of file cdm.h.

10.3.2.10 #define cmdMode ('m')

Command Mode Bits.

Definition at line 253 of file cdm.h.

10.3.2.11 #define cmdOperation ('o')

Command Initiates the canned operations.

Definition at line 257 of file cdm.h.

10.3.2.12 #define cmdPlaceholder ('%')

Command (placeholder for future commands)

Definition at line 259 of file cdm.h.

10.3.2.13 #define cmdReport ('r')

Command Report Format.

Definition at line 250 of file cdm.h.

10.3.2.14 #define cmdSequence ('s')

Command Sequence Number.

Definition at line 243 of file cdm.h.

10.3.2.15 #define cmdSerial ('S')

Command Serial Number.

Definition at line 242 of file cdm.h.

10.3.2.16 #define cmdStyleCode ('h')

Command Hardware Style Code in Hex.

Definition at line 246 of file cdm.h.

10.3.2.17 #define cmdTime ('t')

Command Time.

Definition at line 256 of file cdm.h.

10.3.2.18 #define cmdVersion ('v')

Command Version Number.

Definition at line 244 of file cdm.h.

10.3.2.19 #define delimAssign ('=')

Delimiter for Parameter Value Assignment.

Definition at line 234 of file cdm.h.

10.3.2.20 #define delimCS ('∗')

Delimiter for Checksum.

Definition at line 231 of file cdm.h.

10.3.2.21 #define delimPhrase (';')

Delimiter for Command Phrases.

Definition at line 232 of file cdm.h.

10.3.2.22 #define delimQuery ('?')

Delimiter for Parameter Value Query.

Definition at line 235 of file cdm.h.

10.3.2.23 #define delimQuot1 (0x22)

Delimiter for Quoted Strings - Double Quote '"'.

Definition at line 238 of file cdm.h.

10.3.2.24 #define delimQuot2 (0x27)

Delimiter for Quoted Strings - Single Quote "'".

Definition at line 239 of file cdm.h.

10.3.2.25 #define delimQuot3 (0x60)

Delimiter for Quoted Strings - Accent Grave '‘'.

Definition at line 240 of file cdm.h.

10.3.2.26 #define delimReport (':')

Delimiter for Parameter Value Report.

Definition at line 236 of file cdm.h.

10.3.2.27 #define delimSubParam (',')

Delimiter for Sub-Parameters.

Definition at line 233 of file cdm.h.

10.3.2.28 #define delimTest ('∼')

Delimiter for Parameter Value Test.

Definition at line 237 of file cdm.h.

10.3.2.29 #define SixtyThree (0x5F)

Sixbit character 63 is ASCII '_'.

Definition at line 270 of file cdm.h.

10.3.2.30 #define SixtyTwo (0x24)

Sixbit character 62 is ASCII '$'.

Definition at line 269 of file cdm.h.

10.3.3 Enumeration Type Documentation

10.3.3.1 enum cdState_t

cdState_t Allowed States for the PCD-Bus Interface

Enumerator

cdWaitIdle Wait timeout in progress before becoming Idle.

cdIdle The PCD-Bus is idle waiting for the next message.

cdRecvData Message reception is in progress.

cdTrmtData Message transmission is in progress.

cdRecvDone A completed message is ready to be processed.

cdTrmtDone Transmit was successful.

Definition at line 294 of file cdm.h.

```
{   // Allowed States for the PCD-Bus Interface
    cdWaitIdle,
    cdIdle,
    cdRecvData,
    cdTrmtData,
    cdRecvDone,
    cdTrmtDone
} cdState_t;
```

10.3.3.2 enum uiState_t

uiState_t Allowed States for Buttons or Scroll Wheel

Enumerator

none All buttons are up.

up Up button is pressed.

down Down button is pressed.

enter Enter button is pressed.

back (virtual) Back button was pressed

holding Button is being held down (i.e. auto-repeating)

Definition at line 280 of file cdm.h.

```
{   // **uiState_t** Allowed States for Buttons or Scroll Wheel
    none,
    up,
    down,
    enter,
    back,
    holding
} uiState_t;
```

10.3.4 Function Documentation

10.3.4.1 uint8_t doPhrase (ringSelect_t *ringT*, ringSelect_t *ringF*)

doPhrase Process the single Command that is already isolated in ringF

The command and its parameters are in ringF. Results will be placed in ringT.

Errors can result from unrecognized or malformed commands.

After we return, the caller should verify that 1) There were no characters left unused in ringF, and 2) A string comparison (X∼str) has not set the compareBits.fMismatch bit.

Direct replies to the command will be APPENDED to ringT.

Commands may affect global memory or operating modes, or the contents of other rings, even if an error indication is ultimately returned.

Parameters

ringT	ring to append the reply to
ringF	ring (usually ringSCR) containing the phrase to execute

Returns

value 0 indicates success, 1 means failure

Definition at line 1680 of file cdm.c.

```
{   // **doPhrase** Process the single Command that is already isolated in the Scratch Ring
    uint16_t ch;    // Command Character byte
    uint32_t cp;    // Command parameter
    uint8_t op;     // Command operation byte
    int32_t val;
    uint8_t i;
    ringFrom = ringF;
    ringTo = ringT;
    ch = get();     // Get the command character
    cp = getSix();  // Get the command parameter value
    op = get();
    if ((op != delimAssign) &&
        (op != delimReport) &&
        (op != delimQuery) &&
        (op != delimTest)) return 1;

    if (op == delimTest) ringTo = ringCMP;  // Output to the (virtual) Comparison Ring

    switch (ch) {
        case cmdSerial:   // Command is Serial Number ===================
            switch (op) {
                case delimAssign:   // S=xxx
                        // Disallow Setting the Serial Number more than once
                    if (eeGet32(nvSN) != 0xFFFFFFFF) return 1;
                    EEADR = nvSN;
                    eePut32(getSix());
                    return 0;
                case delimQuery:    // S?
                    putSimpleFieldID(ch);
                case delimTest:     // S~xxx
                    putSix(eeGet32(nvSN));
                    return 0;
                case delimReport:   // S:xxx Sender's Serial Number, use as response destination
                    lastCommandSender = getSix();
                    return 0;
            }
            break;
        case cmdSequence:   // Command is Message Sequence Number =========
            switch (op) {
                case delimAssign:   // s=hh
                    EEADR = nvSeq;
                    eePut(getHex());  // Get the new message sequence number and save it in EEPROM
                    return 0;
                case delimQuery:    // s?
                    putSimpleFieldID(ch);

                    ch = eeGet(nvSeq) + 1;
                    EEADR = nvSeq;      // Increment the message sequence number
                    eePut(ch);         // and save it in EEPROM

                    putHex(ch);     // Put the sequence number in the outgoing message
                    return 0;
                case delimReport:   // s:hh Sender's Sequence Number
                    val = getHex();
                    if (lastCommandSeq == val) return 1;    // Blow off if duplicate
                    lastCommandSeq = val;
                    return 0;
            }
            break;
        case cmdVersion:    // Command is Version Number ==================
            switch (op) {
                case delimAssign:   // v=hh
                        // Disallow Setting the Version for non-manufacturing commands
```

```
                    if (eeGet32(nvSN) != 0xFFFFFFFF) return 1;
                    if (cp == 1) {  // v1=hhhhhh
                        EEADR = nvVer;
                        eePut32(getHex());  // Store the version number in EEPROM
                    }
                    if (cp == 2) {  // v2=hhhhhh
                        EEADR = nvDate;
                        eePut32(getHex());  // Store the version Date in EEPROM
                    }
                    return 0;
                case delimQuery:    // v1?    // v2?
                    putSimpleFieldID(ch);
                case delimTest:     // v1~hhh
                    if (cp == 0) cp = 1;    // version number only without punctuation
                    showVersion(cp);
                    return 0;
            }
            break;
        case cmdID:         // Command is Identification String ===================
            switch (op) {
                case delimAssign:   // I=sss
                    EEADR = nvID;
                    for (i=0; i<16; i++) {
                        if (peek(0) == EOF) {
                            eePut(0);   // Ensure we have a null unless all 16 bytes used;
                            break;
                        }
                        eePut(get());   // will stop at first null
                    }
                    return 0;
                case delimQuery:    // I?
                    putSimpleFieldID(ch);
                case delimTest:     // I~sss
                    EEADR = nvID;
                    for (i=0; i<16; i++) {
                        ch = eeGet(EEADR++);
                        if (ch == 0) break;   // will stop at first null in EEPROM
                        put(ch);
                    }
                    return 0;
            }
            break;
        case cmdHardware:         // Command is Hardware Model String ===================
            switch (op) {
                case delimAssign:   // H=sss
                        // Disallow Setting the Version for non-manufacturing commands
                    if (eeGet32(nvSN) != 0xFFFFFFFF) return 1;
                    EEADR = nvModel;
                    for (i=0; i<16; i++) {
                        if (peek(0) == EOF) {
                            eePut(0);   // Ensure we have a null unless all 16 bytes used;
                            break;
                        }
                        eePut(get());   // will stop at first null
                    }
                    return 0;
                case delimQuery:    // H?
                    putSimpleFieldID(ch);
                case delimTest:     // H~sss
                    EEADR = nvModel;
                    ringFrom = ringEE;
                    for (i=0; i<16; i++) {
                        ch = eeGet(EEADR++);
                        if (ch == 0) break;   // will stop at first null in EEPROM
                        put(ch);
                    }
                    return 0;
            }
            break;
        case cmdStyleCode:         // Command is Hardware Style Code ===================
            switch (op) {
                case delimAssign:
                        // Disallow Setting the Version for non-manufacturing commands
                    if (eeGet32(nvSN) != 0xFFFFFFFF) return 1;
                    EEADR = nvStyle;
                    eePut(getHex());   // Hex Style Code
                    return 0;
                case delimQuery:
                    putSimpleFieldID(ch);
                case delimTest:
                    putHex(eeGet(nvStyle));
```

```
                    return 0;
            }
            break;
        case cmdAnalog:     // Command is Analog Voltage ===================
            switch (op) {
                case delimQuery:
                    if (cp == EOF) return 1;    // Not a valid parameter
                    putIndexedFieldID(ch, cp);
                case delimTest:
                    if (cp == EOF) return 1;    // Not a valid parameter
                    adcShowChannel(cp);
                    return 0;
            }
            break;
        case cmdMode:       // Command is Mode bits ===================
            switch (op) {
                case delimAssign:
                    switch (cp) {
                        case 0:
                            EEADR = nvMode;
                            eePut16(getHex());  // Store the mode bits in EEPROM
                            ModeBits.w = eeGet16(nvMode);   // Immediately change the operating mode
                            return 0;
                        case 1:
                            EEADR = nvSerial;
                            eePut16(getHex());  // Store the mode bits in EEPROM
                            SerialBits.w = eeGet16(nvSerial);   // Immediately change the Serial mode
                            ensureBaud(SerialBits.baud);
                            return 0;
                        case 2:
                            EEADR = nvModbus;
                            eePut8(getHex());  // Store the modbus bits in EEPROM
                            ModbusBits.w = eeGet16(nvModbus);   // Immediately change the operating mode
                            return 0;
                    }
                case delimQuery:
                    putIndexedFieldID(ch, cp);
                case delimTest:
                    switch (cp) {
                        case 0:
                            putHex32(eeGet16(nvMode));
                            return 0;
                        case 1:
                            putHex32(eeGet16(nvSerial));
                            return 0;
                        case 2:
                            putHex32(eeGet(nvModbus));
                            return 0;
                        case 3:
                            putHex32(ErrorBits.w);
                            ErrorBits.w = 0;    // Clear the error bits after reporting
                            return 0;
                    }
            }
            break;
        case cmdMemory:     // Command is Memory Dump ===================
            switch (op) {
                case delimQuery:
                    putSimpleFieldID(ch);
                case delimTest:
                    eeDump(cp * 16, 16);
                    return 0;
            }
            break;
        case cmdDate:       // Command is Date ===================
            switch (op) {
                case delimAssign:
                    val = getInt();
                    rtcSet(val / 10000, (val / 100) % 100, val % 100);
                    return 0;
                case delimQuery:
                    putSimpleFieldID(ch);
                case delimTest:
                    rtcDateTime(cp);
                    return 0;
            }
            break;
        case cmdTime:       // Command is Various Time Values ===================
```

```
            switch (op) {
                case delimAssign:
                    val = getInt();
                    rtcElapsed = (val / 10000)*24*60 + ((val / 100) % 100)*60 + (val % 100);
                            // !!! this is wrong. Never set rtcElapsed this way
                    return 0;
                case delimQuery:
                    putSimpleFieldID(ch);
                case delimTest:
                    rtcDateTime(cp);
                    return 0;
            }
            break;
        case cmdDiag:       // Command is Diagnostic Operations ===================
            switch (op) {
                case delimQuery:
                    if (cp == 0) {  // z0?
                        eeDumpAll(); return 0;
                    }
                    if (cp == 1) {  // z1?
                        putIndexedFieldID(ch, cp);
                        putInt(pollLongest,0); pollLongest=0;
                        return 0;
                    }
                    if (cp == 2) {  // z2?
                        putIndexedFieldID(ch, cp);
                        putHex(cdDiagRB); cdDiagRB = 0; // Received Begin
                        putHex(cdDiagRE); cdDiagRE = 0; // Received Error
                        put(',');
                        putHex(cdDiagTB); cdDiagTB = 0; // Transmitted Begin
                        putHex(cdDiagTE); cdDiagTE = 0; // Transmitted Error
                        put(',');
                        putHex(cdDiagRD); cdDiagRD = 0; // Received Discarded
                        return 0;
                    }
            }
            break;
        case cmdForward:       // Command is Forward to PCD-bus ===================
            switch (op) {
                case delimAssign:
                    if (cp == 0) {  // F0=string to Forward to PCD-bus
                        ringCopy(ringCDtx, ringFrom);
                        return 0;
                    }
                    if (cp == 1) {  // F1=string to Forward to EUSART
                        if (fPower == 0) {
                            sleepLevel(2);   // Turn on aux power
                            wait(2);   // No more than 2 secs
                        }
                        ringCopy(ringTX, ringFrom);
                        return 0;
                    }
                    if (cp == 2) {  // F2=hex string to Forward to EUSART
                        if (fPower == 0) {
                            sleepLevel(2);   // Turn on aux power
                            wait(2);   // No more than 2 secs
                        }
                        ringTo = ringTX;
                        while (peek(1) != EOF) {
                            put( hexNybble(get()) * 16 + hexNybble(get()) );
                        }
                        return 0;
                    }
                    if (cp == 3) {  // F3=string to Forward to Message Ring
                        ringCopy(ringMSG, ringFrom);
                        return 0;
                    }
                case delimQuery:    // F?
                    if (cp == 0) {  // F0? Return Forwarded message without bracketing
                        ringCopy(ringTo, ringMSG);
                        return 0;
                    }
                    if (cp == 1) {  // F1? Return Forwarded message with bracketing
                        putIndexedFieldID(ch,cp);
                        put('"');
                        ringCopy(ringTo, ringMSG);
                        put('"');
                        return 0;
                    }
                    if (cp == 2) {  // F2? Return Forwarded message as a hex string
                        putIndexedFieldID(ch,cp);
```

```
                        ringFrom = ringMSG;
                        while (peek(0) != EOF) putHex(get());
                        return 0;
                    }
            }
            break;
        case cmdReport:       // Command is Report Format ===================
            switch (op) {
                case delimAssign:
                    if (cp == 0) EEADR = nvCmd0; else
                    if (cp == 1) EEADR = nvCmd1; else
                    if (cp == 2) EEADR = nvCmd2; else break;
                    for (i=0;i<32;i++) {
                        ch = get();
                        if (ch == EOF) break;
                        eePut(ch);
                    }
                    return 0;
                case delimQuery:    // r0?
                    putIndexedFieldID(ch, cp);
                    if (cp == 0) EEADR = nvCmd0; else
                    if (cp == 1) EEADR = nvCmd1; else
                    if (cp == 2) EEADR = nvCmd2; else break;
                    put(delimQuot3);
                    for (i=0;i<32;i++) {
                        ch = eePeek(i);
                        if (ch == delimQuot3) put(ch); // Double embedded quotes
                        if (ch == EOF) break;
                        put(ch);
                    }
                    put(delimQuot3);
                    return 0;
            }
            break;
        case cmdDest:       // Command is Report Destination ===================
            switch (op) {
                case delimAssign:
                    if (cp == 0) EEADR = nvDest0; else          // R0=sss
                    if (cp == 1) EEADR = nvDest1; else          // R1=sss
                    if (cp == 2) EEADR = nvDest2; else break;   // R2=sss
                    for (i=0;i<16;i++) {
                        ch = get();
                        if (ch == EOF) {
                            eePut(0);   // Null terminator
                            break;
                        }
                        eePut(ch);
                    }
                    return 0;
                case delimQuery:
                    putIndexedFieldID(ch, cp);
                    if (cp == 0) EEADR = nvDest0; else          // R0?
                    if (cp == 1) EEADR = nvDest1; else          // R1?
                    if (cp == 2) EEADR = nvDest2; else break;   // R2?
                    put(delimQuot3);
                    for (i=0;i<16;i++) {
                        ch = eePeek(i);
                        if (ch == delimQuot3) put(ch); // Double embedded quotes
                        if (ch == 0x00) break;  // EEPROM Strings end with null
                        put(ch);
                    }
                    put(delimQuot3);
                    return 0;
            }
            break;
        case cmdInterval:   // Command is Interval ===================
            switch (op) {
                case delimAssign:   // ip=ii
                    val = getInt();
                    if (cp ==10) {rtcTimerA = val; return 0; }  // iA=ii
                    if (cp ==11) {rtcTimerB = val; return 0; }  // iB=ii
                    if (cp ==12) {rtcTimerC = val; return 0; }  // iC=ii

                    if (cp == 0) {EEADR = nvInt0; rtcTimerA = val; } else
                    if (cp == 1) {EEADR = nvInt1; rtcTimerB = val; } else
                    if (cp == 2) {EEADR = nvInt2; rtcTimerC = val; } else break;
                    eePut32(val);
                    return 0;
                case delimQuery:    // ip?
                    putIndexedFieldID(ch, cp);
                case delimTest:     // ip~sss
```

```
                    if (cp ==10) {putInt(rtcTimerA, 0); return 0; }
                    if (cp ==11) {putInt(rtcTimerB, 0); return 0; }
                    if (cp ==12) {putInt(rtcTimerC, 0); return 0; }

                    if (cp == 0) {EEADR = nvInt0; } else
                    if (cp == 1) {EEADR = nvInt1; } else
                    if (cp == 2) {EEADR = nvInt2; } else break;
                    putInt(eeGet32(EEADR), 0);
                    return 0;
            }
            break;
        case cmdOperation:   // Command is Operation Cycle ===================
            switch (op) {
                case delimAssign:
                    opPhase = cp;       // op=
            }
            break;
        case cmdPlaceholder:   // Command is (placeholder) ===================
            switch (op) {
                case delimAssign:   // cp=v
                    return 0;
                case delimQuery:    // cp?
                    putSimpleFieldID(ch);
                case delimTest:     // cp~v
                    return 0;
                case delimReport:   // cp:v
                    return 0;
            }
            break;
    }
    return 1;   // Unrecognized Command
}
```

10.3.4.2 void doSentence (ringSelect_t *ringT*, ringSelect_t *ringF*)

doSentence Verify the checksum and process the sequence of phrases

The destination ringT is cleared. Consecutive commands are moved from ringF to ringSCR for execution by doPhrase(). If a failure is indicated, the remainder of the Command Sentence is flushed. If all commands are handled sucessfully and the output result is not NULL we append a checksum to the Response. Any error causes a NULL ringT.

Parameters

ringT	The destination Ring Buffer for the Response to the Command
ringF	Selects the Ring Buffer that is the Origin of the Command Sentence

Definition at line 2098 of file cdm.c.

```
{   // **doSentence** Verify the checksum and process the sequence of phrases
    uint16_t ch;
    ringReset(ringT);
    FlagBits.fNeedID = 1;
    while (1) {
        wait(0);    // Be sure to poll....
        ringFrom = ringF;
        ch = peek(0);
        if ((ch == EOF) || (ch == delimCS)) {
            if (FlagBits.fNeedID) break;    // null response message
            ringTo = ringT;
            put(delimCS);               // '*' is included in the checksum
            putHex(checkSum(ringT));    // two hex characters complete the Result Sentence
            ringReset(ringF);
            return;     // This is the successful return
        }
        getPhrase(ringSCR, ringF);  // Copy the next command to the scratch ring
        CompareBits.fMismatch = 0;  // Clear any previous comparison results
//        CompareBits.fAlphaOnly = 1; // Default is to ignore punctuation
        CompareBits.fAlphaOnly = 0; // Default is to expect punctuation
        if (doPhrase(ringT, ringSCR)) break;   // Command error or unrecognized
        if (scrUsed) break;     // Extra stuff on the end of the phrase
        if (CompareBits.fMismatch) break;   // This was a compare command that failed
    }
    ringReset(ringT);   // This is either NULL response or error
```

```
    ringReset(ringF);
}
```

10.3.4.3 void getPhrase (ringSelect_t *ringT*, ringSelect_t *ringF*)

getPhrase Copy the next delimited command phrase from ringFrom to the scrRing

Copy a phrase from ringFrom to ringTo ending with EOF, a ';' or a '∗'.

In order to allow embedded ';' for setting strings into EEPROM, etc. a quoting mechanism is used.

We allow quoted strings bracketed by single-quotes, double-quotes or accent-grave.

Embedded quotes may be included by doubling the quotes within the string.

The beginning and ending quotes are stripped from the result.

The checksum delimiter MAY NOT be included in a quoted string. This ensures compatibility with the Checksum calculation / verification algorithm. Embedded '∗' should not be necessary in any case.

Parameters

ringT	
ringF	

Definition at line 1614 of file cdm.c.

```
{   // **getPhrase** Copy the next delimited command from ringFrom to the scrRing
    uint16_t ch;
    uint8_t  quote = 0;
    ringTo = ringT;
    ringFrom = ringF;
    ringReset(ringTo);
    while (1) {
        ch = peek(0);
        if (ch == EOF) break;
        if (ch == 0x0) break;   // null terminates command read from EEPROM
        if (ch == delimCS) break;       // Expect Checksum on end of command
        get();
        if (quote == 0) {
            if ((ch==0x22) || (ch==0x27) || (ch==0x60)) {
                quote = ch; // Begin a quoted string with one of these symbols
                ch = 0x00;  // Strip the quotes from the phrase
            }
        } else {
            if (ch == quote) {
                    // Doubled quotes within a string become one quote
                if (peek(0) == quote) get(); else {
                    quote = 0x00;   // The end of a matching quote string
                    ch = 0x00;      // Strip the quotes from the phrase
                }
            }
        }
        if (quote == 0) {   // Ignore delimiters within quoted strings
            if (ch == delimPhrase) break;   // Parameter delimiter for end of command
        }
        if (ch > 0x20) put(ch);      // Discard non-printing characters
    }
}
```

10.3.4.4 uint8_t isAlpha (uint16_t *ch*)

isAlpha If the character ch is in the SixBit alphanumeric set return true

Parameters

ch	

Returns

True if ch is in the SixBit alphanumeric set and false otherwise

Definition at line 323 of file cdm.c.

```
324 {   // **isAlpha** If the character ch is in the SixBit alphanumeric set return true
325     if (('0' <= ch) && (ch <= '9')) return 1;
326     if (('A' <= ch) && (ch <= 'Z')) return 1;
327     if (('a' <= ch) && (ch <= 'z')) return 1;
328     if (ch == SixtyTwo) return 1;
329     if (ch == SixtyThree) return 1;
330     return 0;
331 }
```

10.3.4.5 void putFieldDelimiter (void)

putFieldDelimiter Put the field delimiter in the result ring with possible leading sender ID

- A Field Delimiter (';') is placed between fields in a Result Sentence. The first fields in a Result Sentence should normally be the sender's serial number and the recipient's serial number.

All fields in the Response Sentence are potentially optional, so we have no a priori knowlwdge of the need for separators.

We automate this process, and allow for the possibility of null responses by using a global flag bit FlagBits.fNeedID.

Definition at line 1648 of file cdm.c.

```
1649 {   // **putFieldDelimiter** Put the field delimiter in the result ring with possible leading sender ID
1650     if (FlagBits.fNeedID) {
1651         FlagBits.fNeedID = 0;
1652         put('S');
1653         put(delimReport);
1654         putSix(eeGet32(nvSN));  // Sender's Serial number is first in result
1655         if (lastCommandSender) {
1656             put(delimPhrase);    // Delimiter between result phrases
1657             put('r');
1658             put(delimReport);
1659             putSix(lastCommandSender);
1660         }
1661     }
1662     put(delimPhrase);    // Delimiter between result phrases
1663 }
```

10.3.4.6 void putIndexedFieldID (uint8_t *ch*, uint16_t *n*)

putIndexedFieldID Output a properly formatted Field Identifier (like ',a3:')

Parameters

ch	The Field Identifier character

n	The Field index value

Definition at line 1672 of file cdm.c.

```
1673 {   // **putIndexedFieldID** Output a properly formatted Field Identifier (like ';a3:')
1674     putFieldDelimiter();
1675     put(ch);
1676     putSix(n);
1677     put(delimReport);
1678 }
```

10.3.4.7 void putSimpleFieldID (uint8_t *ch*)

putSimpleFieldID Output a properly formatted Field Identifier (like ',S:')

Parameters

ch	The Field Identifier character

Definition at line 1665 of file cdm.c.

```
1666 {   // **putSimpleFieldID** Output a properly formatted Field Identifier (like ';S:')
1667     putFieldDelimiter();
1668     put(ch);
1669     put(delimReport);
1670 }
```

10.3.4.8 void showVersion (uint8_t *n*)

showVersion Show Version and Version Date as selected by parameter bits

0x01 Show the software version 0x02 Show the software version date 0x04 Include '.' and '-' separators (i.e. 1.2.34 and 2014-02-15) otherwise we get 1020034 and 20140215 0x08 Include a single space between version and date

Parameters

n	Bit mask for controlling the output presentation

Definition at line 1586 of file cdm.c.

```
1587 {   // **showVersion** Show Version and Version Date as selected by parameter bits
1588     if (n & 1) {
1589         if (n & 4) {
1590             putHex32(eeGet(nvVer));
1591             put('.');
1592             putHex32(eeGet(nvVer+1));
1593             put('.');
1594             putHex32(eeGet16(nvVer+2));
1595         } else {
1596             putHex32(eeGet32(nvVer));
1597         }
1598     }
1599     if (n & 8) put(' ');
1600     if (n & 2) {
1601         if (n & 4) {
1602             putHex(eeGet(nvDate));
1603             putHex(eeGet(nvDate+1));
1604             put('-');
1605             putHex(eeGet(nvDate+2));
1606             put('-');
1607             putHex(eeGet(nvDate+3));
1608         } else {
1609             putHex32(eeGet32(nvDate));
1610         }
1611     }
1612 }
```

10.3.5 Variable Documentation

10.3.5.1 CompareBits_t CompareBits

Control and results of string comparisons.

Definition at line 352 of file cdm.h.

10.3.5.2 ErrorBits_t ErrorBits

Global Error indication flags.

Definition at line 353 of file cdm.h.

10.3.5.3 FlagBits_t FlagBits

Operating Status Flag Bits.

Definition at line 351 of file cdm.h.

10.3.5.4 uint8_t lastCommandSeq

Definition at line 1411 of file cdm.h.

10.3.5.5 ModbusBits_t ModbusBits

Non-Volatile shadow at nvModbus (8 bits)

ModBus configuration bits. Non-Volatile from nvModbus.

Definition at line 349 of file cdm.h.

10.3.5.6 ModeBits_t ModeBits

Non-Volatile shadow at nvMode (16 bits)

Definition at line 347 of file cdm.h.

10.3.5.7 SerialBits_t SerialBits

Non-Volatile shadow at nvSerial (16 bits)

Definition at line 348 of file cdm.h.

10.4 CDMA Cellular Radio Modem Protocol

Handle SMS Messages using CDMA Hardware.

Functions

- void CDMA (void)

  CDMA *Module has a CDMA Cellular Radio Modem for two-way SMS messages*

10.4.1 Detailed Description

Handle SMS Messages using CDMA Hardware.

10.4.2 Function Documentation

10.4.2.1 void CDMA (void)

CDMA Module has a CDMA Cellular Radio Modem for two-way SMS messages

Definition at line 2743 of file cdm.c.

```
2744 {   // **CDMA** Module has a CDMA Cellular Radio Modem for two-way SMS messages
2745
2746 }
```

10.5 GSM Cellular Radio Modem Protocol

Handle SMS Messages using GSM Hardware.

Functions

- void GSM (void)

  GSM *Module has a GSM Cellular Radio Modem for two-way SMS Messages*
- uint8_t performGSMcycle (uint8_t destN)

  performGSMcycle *Send communication sequence to the GSM radio*
- uint8_t performATOK (void)

  performATOK *Send commands to the radio modem and verify response*
- uint8_t performATCREG (uint8_t ring)

  performATCREG *Send commands to the radio modem and verify registration*
- uint8_t performATCSQ (uint8_t ring)

  performATCSQ *Send commands to the radio modem and get signal quality into ring*
- uint8_t performATCMGF (void)

  performATCMGF *Send commands to the radio modem and set Text Mode Format*
- uint8_t performATCMGS (uint8_t dest, uint8_t ring)

  performATCMGS *Send commands to the radio modem and Send Message from ring*
- uint8_t performATCMGL (uint8_t ringDN, uint8_t ringCMD)

  performATCMGL *Send commands to the radio modem and get inbound commands to ringCMD*
- uint8_t performATCMGD (uint8_t n)

  performATCMGD *Send commands to the radio modem and delete the specified SMS message*

10.5.1 Detailed Description

Handle SMS Messages using GSM Hardware.

10.5.2 Function Documentation

10.5.2.1 void GSM (void)

GSM Module has a GSM Cellular Radio Modem for two-way SMS Messages

Definition at line 2524 of file cdm.c.

```
{   // **GSM** Module has a GSM Cellular Radio Modem for two-way SMS Messages
    sleepLevel(1);  // Aux Power Off
    FlagBits.fAllowCRLF = 1;

    ledAreload = 0x00;  // Slow Flash for idle
    ledBreload = 0x01;

    if (resUsed) {
        if (cdState == cdIdle) {    // Reply with any results to the bus
            ringCopy(ringCDtx, ringRES);
            wait(0);
        }
    } else
    if (opPhase) {
        if (opPhase == 0x0A) performGSMcycle(0);   // Send GSM Message
        if (opPhase == 0x0B) performGSMcycle(1);   // Send GSM Message
        if (opPhase == 0x0C) performGSMcycle(2);   // Send GSM Message
```

```
        opPhase = 0;

        if (resUsed > 0) {  // Radio may return a command to perform
            wait(1);
            if (checkSum(ringRES) <= 0xFF) {
                doSentence(ringNULL, ringRES);   // Execute it
            } else ringReset(ringRES);  // or Discard it
        }
    } else
    if (cdState == cdTrmtDone) cdState = cdWaitIdle; else
    if (cdState == cdRecvDone) {
        ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
        doSentence(ringRES, ringCDrx);// Anytime we see something, try to execute
    it
    } else
    if (rtcTimerA == 0) {
        rtcTimerA = eeGet32(nvInt0);
        EEADR = nvCmd0;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerB == 0) {
        rtcTimerB = eeGet32(nvInt1);
        EEADR = nvCmd1;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerC == 0) {
        rtcTimerC = eeGet32(nvInt2);
        EEADR = nvCmd2;
        doSentence(ringRES, ringEE);
    }
}
```

10.5.2.2 uint8_t performATCMGD (uint8_t *n*)

performATCMGD Send commands to the radio modem and delete the specified SMS message

Parameters

n	

Returns

Definition at line 2328 of file cdm.c.

```
{   // **performATCMGD** Send commands to the radio modem and delete the specified SMS message
    ringReset(ringRX);
    ringFrom = ringRX;
    ringTo = ringTX;
            // Send AT+CMGD=3\n
    putS("AT+CMGD=");   // Command to delete the given inbound message
    putInt(n,0);
    put('\n');
    wait(2);
    if (scan("OK")) return 1;    // Success
    return 0;   // Some kind of failure
}
```

10.5.2.3 uint8_t performATCMGF (void)

performATCMGF Send commands to the radio modem and set Text Mode Format

Returns

Definition at line 2258 of file cdm.c.

```
{   // **performATCMGF** Send commands to the radio modem and Message Format to Text Mode
    ringFrom = ringRX;
    ringTo = ringTX;
    ringReset(ringRX);
    putS ("AT+CMGF=1\n");        // Always Set Text Mode for messages
    wait(1);
    if (scan("OK")) {
        return 1;
    }
    return 0;
}
```

10.5.2.4 uint8_t performATCMGL (uint8_t *ringDN*, uint8_t *ringCMD*)

performATCMGL Send commands to the radio modem and get inbound commands to ringCMD

Parameters

ringDN	
ringCMD	

Returns

Definition at line 2299 of file cdm.c.

```
{   // **performATCMGL** Send commands to the radio modem and get inbound commands to ringCMD
    uint8_t n;  // This will be the message number
    ringReset(ringRX);  // Prepare to talk to the radio
    ringFrom = ringRX;
    ringTo = ringTX;
            // Send AT+CMGL="ALL"\n
    putS("AT+CMGL=\x22" "ALL" "\x22\n");   // Command to list all inbound messages
    wait(2);
    if (scan("L:")) {        // Expect +CMGL: 2,"REC UNREAD","23747",,"..time.." for success
        ringReset(ringDN);
        ringReset(ringCMD);
        if (peek(0) == ' ') get();
        n = getInt();
        ringTo = ringDN;
        scan("\"");
        scan(",\"");  // Next is the sender's directory number
        scanCopy("\"");
        scan("\x0A");
        ringTo = ringCMD;
        scanCopy("\x0D");

        performATCMGD(n);     // Delete the SMS message

        return 1;   // Success;
    }
    return 0;   // Some kind of failure - like no message
}
```

10.5.2.5 uint8_t performATCMGS (uint8_t *dest*, uint8_t *ring*)

performATCMGS Send commands to the radio modem and Send Message from ring

Parameters

ring	

Returns

Definition at line 2271 of file cdm.c.

```
2272 {   // **performATCMGS** Send commands to the radio modem and Send Message from ring
2273     ringFrom = ringRX;
2274     ringTo = ringTX;
2275     ringReset(ringRX);
2276             // Send AT+CMGS="12142323198"\nHello\x1a
2277     ringTo = ringTX;
2278     putS("AT+CMGS=\x22");   // Command to send a text message
2279
2280     EEADR = nvDest0;
2281     if (dest == 1) EEADR = nvDest1;
2282     if (dest == 2) EEADR = nvDest2;
2283     ringCopy(ringTo, ringEE);
2284 //    putS("23747");          // Destination for the message
2285 //    putS("12142323198");    // Destination for the message
2286
2287     putS("\x22\n");         // Closing quote for the destination and a newline
2288     wait(1);
2289     ringCopy(ringTo, ring);
2290     wait(1);
2291     ringReset(ringRX);  // Discard whatever was echoed
2292     putS("\x1a");           // Control-Z on the end of the message
2293     wait(5);
2294     traceDump(4);
2295     if (scan("S:")) return 1;  // Expect "+CMGS: n" for success
2296     return 0;   // Some kind of failure
2297 }
```

10.5.2.6 uint8_t performATCREG (uint8_t *ring*)

performATCREG Send commands to the radio modem and verify registration

Parameters

ring	

Returns

Definition at line 2208 of file cdm.c.

```
2209 {   // **performATCREG** Send commands to the radio modem and verify registration
2210     uint8_t i;
2211     for (i=1; i<20; i++) {
2212         ringFrom = ringRX;
2213         ringTo = ringTX;
2214         // Send AT+CREG?\n and expect either +CREG: 0,1\n or +CREG: 0,5\n
2215         ringReset(ringFrom);
2216         ringTo = ringTX;
2217         putS ("AT+CREG=2\n");        // Check network registration
2218         wait(1);
2219
2220         // Send AT+CREG?\n and expect either
2221         //      +CREG: 0,1,"aaaa","iiii"\n or
2222         //      +CREG: 0,5,"aaaa","iiii"\n
2223         ringReset(ringFrom);
2224         ringTo = ringTX;
```

```
        putS ("AT+CREG?\n");        // Check network registration
        wait(1);

        traceDump(2);
        scan("G: ");
        ringTo = ring;
        ringReset(ringTo);  // The destination ring gets 0,1,"aaaa","iiii"
        scanCopy("\x0D");
        ringFrom = ring;
        if (pos(",1") != EOF) return 1;    // Success
        if (pos(",5") != EOF) return 1;    // Success
        wait(10);
    }
    return 0;   // Failure to Register
}
```

10.5.2.7 uint8_t performATCSQ (uint8_t *ring*)

performATCSQ Send commands to the radio modem and get signal quality into ring

Parameters

ring	

Returns

Definition at line 2241 of file cdm.c.

```
{   // **performATCSQ** Send commands to the radio modem and get signal quality into ring
    ringFrom = ringRX;
    ringTo = ringTX;
    ringReset(ringRX);
    ringReset(ring);
    putS ("AT+CSQ\n");        // Check Signal Quality
    wait(1);
    traceDump(3);
    if (scan("Q: ")) {
        ringTo = ring;
        scanCopy("\0x0D");     // The destination ring gets 15,0
        return 1;
    }
    return 0;
}
```

10.5.2.8 uint8_t performATOK (void)

performATOK Send commands to the radio modem and verify response

Returns

Definition at line 2193 of file cdm.c.

```
{   // **performATOK** Send commands to the radio modem and verify response
    uint8_t i;
    ringFrom = ringRX;
    ringTo = ringTX;
    for (i=1;i<10;i++) {
            // Send AT\n and expect OK\n
        ringReset(ringFrom);
        putS ("AT\n");
```

```
        wait(1);
        if (scan("OK")) return 1;   // Success
    }
    return 0;   // Failure to communicate
}
```

10.5.2.9 uint8_t performGSMcycle (uint8_t *destN*)

performGSMcycle Send communication sequence to the GSM radio

Parameters

destN	

Returns

Definition at line 2127 of file cdm.c.

```
{   // **performGSMcycle** Send communication sequence to the GSM radio
    uint8_t i;
    sleepLevel(2);
    ensureBaud(baud115200);    // This is Radio Baud Rate
    SerialBits.fEchoRx = 0;    // We must not echo stuff received from the Radio

    performATOK();      // Verify communication with modem

    performATCREG(ringSCR);    // Verify Registration

    performATCSQ(ringSCR);     // Verify Signal Quality

    performATCMGF();    // Set Text Mode Formatting for SMS Messages

    ringTo = ringCDtx;
    if (cdState == cdTrmtDone) cdState = cdWaitIdle;
    if (cdState == cdRecvDone) cdState = cdWaitIdle;
    putS("o1=");    // Put a blind command to the Laser module and wait for the results
    for (i=0; i<10; i++) {
        wait(1);
        if (cdState == cdTrmtDone) cdState = cdWaitIdle;
        if (cdState == cdRecvDone) break;   // Got the response from the laser
    }
    ringTo = ringSCR;
    ringReset(ringSCR);
    putS("S:");
    putSix(eeGet32(nvSN));
    putS(";s:");
    i = eeGet(nvSeq) + 1;
    EEADR = nvSeq;      // Increment the message sequence number
    eePut(i);         // and save it in EEPROM
    putHex(i);
    putS(";a0:");
    adcShowChannel(0);
    putS(";i0:");
    putInt(eeGet32(nvInt0), 0);
//    putS(";db:");
//    rtcDateTime(0x25);  // Time as nndnnns
    putS(";d9:");
    rtcDateTime(9);  // Time as nndhhnnss
    putS(";");

            // ringCDrx should be what the laser said
    ringCopy(ringTo, ringCDrx);     // This releases the CD-Bus

    put('*');    // A message
    putHex(checkSum(ringSCR));

    performATCMGS(destN, ringSCR);

    wait(10);   // Add a delay here to help get pending messages

```

```
        // Return only the most recent message
    while (performATCMGL(ringSCR, ringRES)) continue;

    ringFrom = ringRES;
    traceDump(8);

    ringTo = ringTX;
    putS("AT+CREG=0\n");    // sign off and wait for it to de-register
    wait(5);
    sleepLevel(1);  // Turn off the radio when we are done
    return 1;  // Success
}
```

10.6 Non-volatile memory allocation

Memory allocation for non-volatile memory.

Data Structures

- struct ModeBits_t

  ModeBits_t *Universal Controller Operating Mode Control*

Macros

- #define nvVer (0x00)

  32-Bit Firmware Version Number

Enumerations

- enum style_t {
  styleVirgin, styleUser, styleLaser, styleCDMA,
  styleGSM, styleUSB, styleBlue, styleWiFi,
  styleModbus, styleSNAP, styleTest, styleSerial,
  styleRS485 }

  style_t *Board Configuration Style*

Functions

- void copyFromEEs (uint8_t addr, uint8_t max)

  copyFromEEs *Copy a null-terminated string from EEPROM to a Ring Buffer*
- uint8_t eeGet (uint8_t addr)

  eeGet *Set the EEADR and Read character from EEPROM*
- uint8_t eePeek (uint8_t n)

  eePeek *Read character from EEPROM relative to the current EEADR*
- uint16_t eePut (uint16_t ch)

  eePut *Write the character ch to the current EEADR and increment the address*
- uint16_t eeGet16 (uint8_t addr)

  eeGet16 *Get an unsigned 16-bit value from EEPROM*
- uint32_t eeGet32 (uint8_t addr)

  eeGet32 *Get an unsigned 32-bit value from EEPROM*
- void eePut8 (uint16_t val)

  eePut8 *Write the byte value to the current EEADR and increment the address*
- void eePut16 (uint16_t val)

  eePut16 *Write the unsigned 16-bit value to the current EEADR and increment the address*
- void eePut32 (uint32_t val)

  eePut32 *Write the unsigned 32-bit value to the current EEADR and increment the address*
- void eeDump (uint8_t addr, uint8_t n)

  eeDump *Dump a row of bytes from EEPROM to the Ring Buffer ringTo.*
- void eeDumpAll (void)

eeDumpAll *Make a user-friendly dump of all of EEPROM to ringTo*

- void eeInit ()

  eeInit *Initialize the EEPROM just after manufacturing since the compiler can't do it*

10.6.1 Detailed Description

Memory allocation for non-volatile memory.

All devices implement a number of operating parameters stored in non-volatile memory.

Here we define the non-volatile EEPROM Addresses Address Address prefix is nv, function prefix is ee

10.6.2 Macro Definition Documentation

10.6.2.1 #define nvVer (0x00)

32-Bit Firmware Version Number

Definition at line 1002 of file cdm.h.

10.6.3 Enumeration Type Documentation

10.6.3.1 enum style_t

style_t Board Configuration Style

Style indicates the particular hardware associated with this Universal Controller.

This is a non-volatile parameter obtained from EEPROM location nvStyle.

Enumerator

styleVirgin 0: Device has never been initialized

styleUser 1: Device has user interface hardware

styleLaser 2: Device is a Laser Distance Measurement Module

styleCDMA 3: Device is a CDMA Radio Communication Link

styleGSM 4: Device is a GSM Radio Communication Link

styleUSB 5: Device is a USB Interface Module

styleBlue 6: Device is a Bluetooth Radio Communication Link

styleWiFi 7: Device is a WiFi Radio Communication Link

styleModbus 8: Device is a ModBus Interface Module

styleSNAP 9: Device is the Serial Number Assignment Processor for manufacturing

styleTest A: Device is a development test unit.

styleSerial B: Device is the Serial Interface Unit.

styleRS485 C: Device is an RS-485 serial unit.

Definition at line 163 of file cdm.h.

```
{   // **style_t** Board Configuration Style is Hex Byte
    styleVirgin,
    styleUser,
    styleLaser,

    styleCDMA,
    styleGSM,
    styleUSB,
    styleBlue,
    styleWiFi,

    styleModbus,

    styleSNAP,
    styleTest,
    styleSerial,
    styleRS485
} style_t;
```

10.6.4 Function Documentation

10.6.4.1 void copyFromEEs (uint8_t *addr*, uint8_t *max*)

copyFromEEs Copy a null-terminated string from EEPROM to a Ring Buffer

Copy a string of up to max characters from EEPROM starting at addr to ringTo. The string may terminate with a null character. The null is not copied.

Parameters

addr	EEPROM Starting Address
max	Maximum number of characters to copy

Definition at line 174 of file cdm.c.

```
{   // **copyFromEEs** Copy a null-terminated string from EEPROM
    uint8_t ch;
    uint8_t i;
    for (i = 0; i < max; i++) {    // Explicitly limit it to max bytes
        ch = eeGet(addr+i);
        if (ch == 0) break;
        put(ch);
    }
    EEADR = addr;   // Explicitly leave the EEPROM address back at the beginning
}
```

10.6.4.2 void eeDump (uint8_t *addr*, uint8_t *n*)

eeDump Dump a row of bytes from EEPROM to the Ring Buffer ringTo.

Send a single Dump line indicating the Starting Address, n bytes in Hex and the same n bytes as printable ASCII.

Parameters

addr	Starting EEPROM Address for dump
n	Number of bytes to dump in this row

Definition at line 796 of file cdm.c.

```
{   // **eeDump** Dump n bytes to the current output ring
    uint8_t i;
    putHex(addr);  put(0x20);  put('-');
    EEADR = addr;   // Set the EEPROM base Read Address
    for (i=0; i<n; i++) {if (i == 8) put(' '); put(' '); putHex(
eePeek(i)); }
    put(' ');
```

```
    for (i=0; i<n; i++) {if (i == 8) put(' '); putDumpChar(eePeek(i)); }
}
```

10.6.4.3 void eeDumpAll (void)

eeDumpAll Make a user-friendly dump of all of EEPROM to ringTo

All of EEPROM will be dumped in lines of 16-bytes each.

Definition at line 806 of file cdm.c.

```
{   // **eeDumpAll** Make a user-friendly dump of all of EEPROM to ringTo
    uint8_t a;
    ringTo = ringTX;    // Dump to UART Serial Output
    for (a = 0; a < 16; a++) {
        putCRLF();
        eeDump(a*16, 16);
    }
    putCRLF();
}
```

10.6.4.4 uint8_t eeGet (uint8_t *addr*)

eeGet Set the EEADR and Read character from EEPROM

Set the given address into EEADR.

Read the byte value from the EEPROM location.

Increment EEADR.

Parameters

addr	EEPROM Address to read

Returns

Byte value read from EEPROM

Definition at line 686 of file cdm.c.

```
{   // **eeGet** Set the EEADR and Read character from EEPROM
    EEADR = addr;
    EECON1bits.EEPGD = 0;   // Access EEPROM, not program flash, and
    EECON1bits.CFGS = 0;    // Not the config bits
    EECON1bits.RD = 1;      // Do the read
    return EEDATA;
}
```

10.6.4.5 uint16_t eeGet16 (uint8_t *addr*)

eeGet16 Get an unsigned 16-bit value from EEPROM

Parameters

addr	

Returns

Definition at line 734 of file cdm.c.

```
735 {   // **eeGet16** Get an unsigned 16-bit value from EEPROM
736     uint16_t res;
737     res = eeGet(addr);
738     res = (res << 8) + eeGet(addr+1);
739     EEADR = addr;           // Restore the base EEPROM Address
740     return res;
741 }
```

10.6.4.6 uint32_t eeGet32 (uint8_t *addr*)

eeGet32 Get an unsigned 32-bit value from EEPROM

Parameters

addr	

Returns

Definition at line 743 of file cdm.c.

```
744 {   // **eeGet32** Get an unsigned 32-bit value from EEPROM
745     uint32_t res;
746     res = eeGet(addr);
747     res = (res << 8) + eeGet(addr+1);
748     res = (res << 8) + eeGet(addr+2);
749     res = (res << 8) + eeGet(addr+3);
750     EEADR = addr;           // Restore the base EEPROM Address
751     return res;
752 }
```

10.6.4.7 void eeInit ( )

eeInit Initialize the EEPROM just after manufacturing since the compiler can't do it

Definition at line 2899 of file cdm.c.

```
2900 {   // **eeInit** Initialize the EEPROM just after manufacturing since the compiler can't do it
2901     if (eeGet(nvID) == 0xFF) {  // Only do this when we are freshly programmed
2902         ringTo = ringEE;
2903         EEADR = nvID;       putS("Virgin"); eePut(0);
2904         EEADR = nvModel;    putS("N/A"); eePut(0);
2905         EEADR = nvDest0;    putS("23747"); eePut(0);       // Aeris server
2906         EEADR = nvDest1;    putS("12142323198"); eePut(0); // Brian's Cell Phone
2907         EEADR = nvDest2;    eePut(0);
2908         EEADR = nvCmd0;     putS("oA="); eePut(0);
2909         EEADR = nvCmd1;     putS("oB="); eePut(0);
2910         EEADR = nvCmd2;     eePut(0);
2911 //        EEADR = nvInt0; eePut32(15*60);     // every 15 minutes
2912 //        EEADR = nvInt1; eePut32(60*60);     // every Hour
2913 //        EEADR = nvInt0; eePut32(5*60);      // Server Every 5 minutes
2914
2915         EEADR = nvVer;      eePut32(versionNumber);
```

```
        EEADR = nvDate;     eePut32(versionDate);
        EEADR = nvStyle;    eePut(styleVirgin);

        EEADR = nvMode;     eePut16(0);

        SerialBits.w = 0;
        SerialBits.baud = baud9600;
        SerialBits.fEchoRx = 1;
        EEADR = nvSerial;   eePut16(SerialBits.w);

        EEADR = nvModbus;   eePut(0);
    }

        // Load all Non-Volatile Shadow Values into RAM
    ModeBits.w = eeGet16(nvMode);
    SerialBits.w = eeGet16(nvSerial);
    ModbusBits.w = eeGet(nvModbus);

    rtcTimerA = eeGet32(nvInt0);
    rtcTimerB = eeGet32(nvInt1);
    rtcTimerC = eeGet32(nvInt2);
}
```

10.6.4.8 uint8_t eePeek (uint8_t *n*)

eePeek Read character from EEPROM relative to the current EEADR

Read a byte from EEPROM at an address offset from the current EEADR.

EEADR Is not modified.

The EEPROM Address space is 256 bytes - addresses will wrap around.

Parameters

n	Address offset from current EEADR to read

Returns

Byte value read from EEPROM

Definition at line 695 of file cdm.c.

```
{   // **eePeek** Read character from EEPROM relative to the current EEADR
    uint8_t addr = EEADR;
    EEADR += n;             // Peek at the EEPROM address offset by n
    EECON1bits.EEPGD = 0;   // Access EEPROM, not program flash, and
    EECON1bits.CFGS = 0;    // Not the config bits
    EECON1bits.RD = 1;      // Do the read
    EEADR = addr;           // Restore the base EEPROM Address
    return EEDATA;
}
```

10.6.4.9 uint16_t eePut (uint16_t *ch*)

eePut Write the character ch to the current EEADR and increment the address

We expect the EEADR to already be set to the target address.

If ch is EOF, we do nothing.

Otherwise, we write the byte value ch to the EEPROM and increment EEADR.

We handle the possible interrupt Enable/Disable situations and ensure that the hardware-imposed write safety conditions are met.

We always wait for completion of the write before return.

We do not perform a read-after-write verify.

Parameters

ch	Byte value to write, or EOF to skip.

Returns

Return value is the same as the input ch

Definition at line 706 of file cdm.c.

```
{   // **ePut** Write the character ch to the current EEADR and increment the address
    if (ch != EOF) {
        EECON1bits.EEPGD = 0;   // Access EEPROM, not program flash, and
        EECON1bits.CFGS = 0;    // Not the config bits
        EECON1bits.RD = 1;      // Read the byte
        if (EEDATA != ch) {     // Only write changes
            EECON1bits.WREN = 1;    // Enable Writes
            EEDATA = ch;            // The data byte to write
            if (INTCONbits.GIE) {   // Interrupts were enabled, we must stop and then restore
                INTCONbits.GIE = 0;
                EECON2 = 0x55;
                EECON2 = 0xAA;
                EECON1bits.WR = 1;  // Start the actual write
                INTCONbits.GIE = 1; // Interrupts back on again
            } else {
                EECON2 = 0x55;
                EECON2 = 0xAA;
                EECON1bits.WR = 1;  // Start the actual write
            }
            while (EECON1bits.WR) continue;  // Wait for the write to complete before returning
            EECON1bits.WREN = 0;    // Disable Writes outside of this routine
        }
        EEADR++;                // Increment the EEPROM Address
    }
    return ch;
}
```

10.6.4.10 void eePut16 (uint16_t *val*)

eePut16 Write the unsigned 16-bit value to the current EEADR and increment the address

Parameters

val	

Definition at line 759 of file cdm.c.

```
{   // **ePut16** Write the unsigned 16-bit value to the current EEADR and increment the address
    ePut((val >> 8) & 0xFF);
    ePut(val & 0xFF);
}
```

10.6.4.11 void ePut32 (uint32_t *val*)

ePut32 Write the unsigned 32-bit value to the current EEADR and increment the address

Parameters

val	

Definition at line 765 of file cdm.c.

```
766 {   // **eePut32** Write the unsigned 32-bit value to the current EEADR and increment the address
767     eePut((val >> 24) & 0xFF);
768     eePut((val >> 16) & 0xFF);
769     eePut((val >> 8) & 0xFF);
770     eePut(val & 0xFF);
771 }
```

10.6.4.12 void eePut8 (uint16_t *val*)

eePut8 Write the byte value to the current EEADR and increment the address

Parameters

val	

Definition at line 754 of file cdm.c.

```
755 {   // **eePut16** Write the unsigned 16-bit value to the current EEADR and increment the address
756     eePut(val & 0xFF);
757 }
```

10.7 Human-readable Input and Output Formatting

Input and output conversion for ring buffers.

Functions

- void putCRLF (void)

  putCRLF *Conditionally send a CR/LF pair to ringTo*
- void putS (const uint8_t *str)

  putS *Put Null-terminated String to ringTo*
- uint16_t putHex (uint16_t ch)

  putHex *Two hex characters or nothing if EOF*
- void putHex32 (uint32_t v)

  putHex32 *Hex value with leading zeroes suppressed*
- void putInt (int32_t v, uint8_t d)

  putInt *Signed integer with optional leading zeroes to ringTo*
- uint8_t sixNybble (uint8_t v)

  sixNybble *Convert six bits into a printable sixBit character*
- void putSix (uint32_t v)

  putSix *Integer Output in Sixbit without leading '0' or '_'*
- uint32_t getSix (void)

  getSix *Get variable-length Sixbit value from selected ring*
- uint16_t hexNybble (uint16_t ch)

  hexNybble *Convert hexadecimal nybble value*
- uint32_t getHex (void)

  getHex *Get variable length hexadecimal value from ringFrom*
- int32_t getInt (void)

  getInt *Get decimal integer from ringFrom*

- #define SixtyTwo (0x24)

  Sixbit character 62 is ASCII '$'.

10.7.1 Detailed Description

Input and output conversion for ring buffers.

10.7.2 Macro Definition Documentation

10.7.2.1 #define SixtyTwo (0x24)

Sixbit character 62 is ASCII '$'.

Definition at line 269 of file cdm.h.

10.7.3 Function Documentation

10.7.3.1 uint32_t getHex (void)

getHex Get variable length hexadecimal value from ringFrom

Returns

Definition at line 658 of file cdm.c.

```
659 {   // **getHex** Get variable length hexadecimal value from ringFrom
660     uint32_t res = 0;
661     uint16_t ch;
662     while ((ch = hexNybble(peek(0))) != EOF) {
663         res = 16 * res + ch;
664         get();
665     }
666     return res;
667 }
```

10.7.3.2 int32_t getInt (void)

getInt Get decimal integer from ringFrom

Returns

Definition at line 669 of file cdm.c.

```
670 {   // **getInt** Get decimal integer from ringFrom
671     int32_t res = 0;
672     uint16_t ch = peek(0);
673     uint8_t  sign = 0;
674     if (ch == '+') {sign = 0; get(); }
675     if (ch == '-') {sign = 1; get(); }
676     while (1) {
677         ch = peek(0);
678         if (ch == 'x') {get(); res = getHex(); break; }
679         if ((0x30 <= ch) && (ch <= 0x39)) res = res * 10 + ch - 0x30; else break;
680         get();
681     }
682     if (sign == 1) res = 0 - res;
683     return res;
684 }
```

10.7.3.3 uint32_t getSix (void)

getSix Get variable-length Sixbit value from selected ring

Returns

Definition at line 631 of file cdm.c.

```
{   // **getSix** Get variable-length Sixbit value from selected ring
    uint32_t res = 0;
    uint16_t  ch;
    if (peek(0) == 0x23) res = 0xFFFFFFFF;  // Sign-extend anything starting with '_'
    while (1) {
        ch = peek(0);
        if ((0x30 <= ch) && (ch <= 0x39)) res = res * 64 + ch - 0x30; else
        if ((0x41 <= ch) && (ch <= 0x5A)) res = res * 64 + ch - 0x41 + 10; else
        if ((0x61 <= ch) && (ch <= 0x7A)) res = res * 64 + ch - 0x61 + 36; else
        if (ch == SixtyTwo) res = res * 64 + 0xFE; else     // '$' for sixbit 0x3E
        if (ch == SixtyThree) res = res * 64 + 0xFF; else   // '_' for sixbit 0x3F
            // For liberal, historical, reasons we also allow '#' for 0x3F
        if (ch == 0x23) res = res * 64 + 0xFF; else break;  // '#' for sixbit 0x3F
        get();
    }
    return res;
}
```

10.7.3.4 uint16_t hexNybble (uint16_t *ch*)

hexNybble Convert hexadecimal nybble value

Parameters

ch	

Returns

Definition at line 650 of file cdm.c.

```
{   // **hexNybble** Convert hexadecimal nybble value
    if ((0x30 <= ch) && (ch <= 0x39)) return ch - 0x30;
    if ((0x41 <= ch) && (ch <= 0x46)) return ch - 0x41 + 10;
    if ((0x61 <= ch) && (ch <= 0x66)) return ch - 0x61 + 10;
    return EOF;
}
```

10.7.3.5 void putCRLF (void)

putCRLF Conditionally send a CR/LF pair to ringTo

Definition at line 503 of file cdm.c.

```
{   // **putCRLF** Send a CR/LF to the ringTo ring with suppression of consecutive CR/LF pairs
    if (FlagBits.fAllowCRLF) {  // Suppresses consecutive CR/LF pairs
        put(0x0D); put(0x0A);   // Iff they are done with this function
        FlagBits.fAllowCRLF = 0;
    }
}
```

10.7.3.6 uint16_t putHex (uint16_t *ch*)

putHex Two hex characters or nothing if EOF

Parameters

ch	

Returns

Definition at line 518 of file cdm.c.

```
519 {   // **putHex** Two hex characters or nothing if EOF
520     if (ch != EOF) {
521         ch = ch & 0xFF;
522         if ((ch & 0xF0) >= 0xA0) put((ch >> 4) - 10 + 'A'); else put((ch >> 4) + '0');
523         if ((ch & 0x0F) >= 0x0A) put((ch & 0x0F) - 10 + 'A'); else put((ch & 0x0F) + '0');
524     }
525     return ch;
526 }
```

10.7.3.7 void putHex32 (uint32_t *v*)

putHex32 Hex value with leading zeroes suppressed

Parameters

v	

Definition at line 528 of file cdm.c.

```
529 {   // **putHex32** Hex value with leading zeroes suppressed
530     FlagBits.fZeroSupp = 1;
531     putHex((v >> 24) & 0xFF);
532     putHex((v >> 16) & 0xFF);
533     putHex((v >>  8) & 0xFF);
534     putHex((v      ) & 0xFF);
535     if (FlagBits.fZeroSupp) {
536         FlagBits.fZeroSupp = 0;
537         put('0');
538     }
539 }
```

10.7.3.8 void putInt (int32_t *v*, uint8_t *d*)

putInt Signed integer with optional leading zeroes to ringTo

Parameters

v	
d	

Definition at line 541 of file cdm.c.

```
542 {   // **putInt** Signed integer with optional leading zeroes to ringTo
543     if (v < 0) FlagBits.fSign = 1;
544     if (FlagBits.fSign) {
545         if (FlagBits.fZeroSupp) {
546             if (v < 0) put('-'); else put('+');
547             FlagBits.fZeroSupp = 1;     // Preserve the zero suppression option
548         } else {
549             if (v < 0) put('-'); else put('+');
550         }
551         FlagBits.fSign = 0;     // Clear the forced sign bit each time
552     }
```

```
553     if (v < 0) v = 0 - v;
554     if (d == 0) {           // Zero width means figure out the required width
555         d = 1;
556         if (v >= 10) d = 2;
557         if (v >= 100) d = 3;
558         if (v >= 1000) d = 4;
559         if (v >= 10000) d = 5;
560         if (v >= 100000) d = 6;
561     }
562     if (d >= 6) put('0' + (v / 100000) % 10); // output exactly d digits
563     if (d >= 5) put('0' + (v / 10000) % 10);
564     if (d >= 4) put('0' + (v / 1000) % 10);
565     if (d >= 3) put('0' + (v / 100) % 10);
566     if (d >= 2) put('0' + (v / 10) % 10);
567     FlagBits.fZeroSupp = 0;
568     put('0' + (v / 1) % 10);
569 }
```

10.7.3.9 void putS (const uint8_t * *str*)

putS Put Null-terminated String to ringTo

Parameters

**str*	Null-terminated character string to send to ringTo

Definition at line 511 of file cdm.c.

```
512 {   // **putS** Put Null-terminated String to ringTo
513     uint8_t i;
514     uint8_t ch;
515     for (i=0; (ch = str[i]) != 0; i++) put(ch);
516 }
```

10.7.3.10 void putSix (uint32_t *v*)

putSix Integer Output in Sixbit without leading '0' or '_'

Parameters

v	

Definition at line 581 of file cdm.c.

```
582 {   //  **putSix** Integer Output in Sixbit without leading '0' or '_'
583     uint8_t ch;
584     int8_t i;
585     if (v & 0x80000000) {   // arrange to sign extend the 'negative' value
586         FlagBits.fSixSupp = 1; // (means suppress '_' for now)
587         FlagBits.fZeroSupp = 0;
588     } else {
589         FlagBits.fSixSupp = 0;
590         FlagBits.fZeroSupp = 1; // (means suppress '0' for now)
591     }
592     for (i=5; i>=0; i--) {
593         if (i == 5) {
594             ch = (v >> 30) & 3;
595             if (ch >= 2) ch |= 0x3C;    // Sign extend 'negative' values
596             ch = sixNybble(ch);
597         } else {
598             ch = sixNybble(v >> (6*i));
599         }
600         if (FlagBits.fSixSupp) {
601             if (ch != SixtyThree) {
602                 FlagBits.fSixSupp = 0;
603                 put(SixtyThree);   // Leading '_' indicates negative value
604                     // Put the first character in a negative number
605                 put(ch);
```

```
            }
        } else
        if (FlagBits.fZeroSupp) {
            if (ch != '0') {
                FlagBits.fZeroSupp = 0;
                     // put a leading zero if it would be mistaken for negative
                if (ch == SixtyThree) put('0');
                    // Put the first character in a positive number
                put(ch);
            }
        } else {
                // Put subsequent characters in the value
            put(ch);
        }
    }
    if (FlagBits.fSixSupp) {
        FlagBits.fSixSupp = 0;
        put(SixtyThree);   // Single '_' indicates -1 value
    }
    if (FlagBits.fZeroSupp) {
        FlagBits.fZeroSupp = 0;
        put('0');   // Single '0' indicates zero value
    }
}
```

10.7.3.11 uint8_t sixNybble (uint8_t *v*)

sixNybble Convert six bits into a printable sixBit character

Parameters

v	

Returns

Definition at line 571 of file cdm.c.

```
{   // **sixNybble** Convert six bits into a printable sixBit character
    v = v & 0x3F;
    if (v < 10) return (0x30 + v);
    if (v < 36) return (0x41 - 10 + v);
    if (v < 62) return (0x61 - 36 + v);
    if (v == 62) return SixtyTwo;   // '$' for sixbit 0x3E
    return SixtyThree;              // '_' for sixbit 0x3F
}
```

10.8 User Interface Menu

Menu for the LCD Display Interface.

Functions

- void menuStep (void)

  menuStep *...*
- void menu (void)

  menu *Handle operation of the Menu state machine*

Variables

- uint8_t menuState

  Index for the Menu Tree State Machine.

10.8.1 Detailed Description

Menu for the LCD Display Interface.

10.8.2 Function Documentation

10.8.2.1 void menu (void)

menu Handle operation of the Menu state machine

Definition at line 2826 of file cdm.c.

```
{   // **menu** Handle operation of the Menu state machine
    if (!ModeBits.fUI) return;
    if (uiState == back) menuState = 0;
    while (1) {
        ringTo = ringDisp; diCursor = 16;
        switch (menuState) {
            case 0xFF:
                menuState = 3;       // wrap around the main menu
                return;

            case 0:
                ModeBits.fUIclock = 1;
                putS("Menu 0 ");
                if (uiState == enter) {menuState = 10; return; }
                menuStep();  // Step up or down in the menu
                return;
            case 1:
                putS("Say Hello ");
                if (uiState == enter) {menuState = 20; return; }
                menuStep();  // Step up or down in the menu
                return;
            case 2:
                putS("Menu 2 ");
                menuStep();  // Step up or down in the menu
                return;
            case 3:
                putS("GSM Send ");
                if (uiState == enter) {menuState = 30; return; }
                menuStep();  // Step up or down in the menu
                return;

            case 10:
                putS("set");
```

```
2860                FlagBits.fSign = 1; putInt(uiScroll,0);
2861                diBlankField(23);
2862                ModeBits.fBlink = 1;
2863                ModeBits.fUnder = 1;
2864                uiCursor = diCursor - 1;    // Blink the last digit of the entry
2865                if (uiState == enter) {uiCursor = 0xFF;
      menuState = 0; uiScroll = 0; }
2866                return;
2867
2868            case 20:
2869                ModeBits.fUIclock = 0;  // Turn off the clock
2870                putS("saying..");
2871                ringTo = ringCDtx;
2872                putS("Hello");
2873                menuState = 21;
2874                return;
2875
2876            case 21:
2877                putS("said.");
2878                diBlankField(23);
2879                if (uiState == enter) {menuState = 0;
      uiScroll = 0; }
2880                return;
2881
2882            case 30:
2883                putS("GSM Sending...");
2884                diBlankField(31);
2885                //GSMsendMessage();
2886                menuState = 31;
2887                return;
2888
2889            case 31:
2890                putS("GSM Done...");
2891                diBlankField(31);
2892                if (uiState == enter) {menuState = 0;
      uiScroll = 0; }
2893                return;
2894        }
2895        menuState = 0;   // Make any illegal state valid and go to top level
2896    }
2897 }
```

10.8.2.2 void menuStep (void)

menuStep ...

// End Menu

Definition at line 2819 of file cdm.c.

```
2820 {   // **menuStep** ...
2821     if (uiScroll > 0) menuState++;
2822     if (uiScroll < 0) menuState--;
2823     uiScroll = 0;
2824 }
```

10.8.3 Variable Documentation

10.8.3.1 uint8_t menuState

Index for the Menu Tree State Machine.

Definition at line 1507 of file cdm.h.

10.9 Support Functions

Various support utilities.

Functions

- uint16_t putDumpChar (uint16_t ch)

  putDumpChar *Send the given character to the ringTo Ring Buffer as a printable form*
- uint16_t putCtrlChar (uint16_t ch)

  putCtrlChar *Send the given character to the ringTo Ring Buffer as a printable form*
- uint16_t checkSum (ringSelect_t ring)

  checkSum *Verify a NMEA-style Checksum of bytes in ring*
- void traceDump (uint8_t index)

  traceDump *Send a copy of whatever is in ringFrom to the PCD-bus as a Znn: message*

10.9.1 Detailed Description

Various support utilities.

10.9.2 Function Documentation

10.9.2.1 uint16_t checkSum (ringSelect_t *ring*)

checkSum Verify a NMEA-style Checksum of bytes in ring

All bytes in the speciified ring up to and including the '∗' are summed.

Following the '∗', expect two ASCII hex characters representing the CRC Big Endian. If these two characters represent a value that matches the expected checksum, return the checksum value.

If the checksum is present but invalid in any way, return EOF.

If the Sentence ends with the checksum delimiter '∗', return the value that should be there plus 0x0100. This allows us to use checkSum() for outbound Sentences and still catch all invalid input.

For testing we specifically allow a single '∗' in place of the two hex characters to be treated as a valid checksum.

Thus, three result cases: 0x0000 - 0x00FF Valid inbound Sentence with valid checksum 0x0100 - 0x01FF Valid outbound Sentence ready to have ASCII hex appended 0xFFFF Anything else ==> error

Parameters

ring	ring to peek from

Returns

Checksum of ring's bytes up to and including the '∗'

Definition at line 885 of file cdm.c.

```
886 {   // **checkSum** Validate a NMEA-style Checksum of bytes in ring
887     uint8_t saveFrom = ringFrom;
888     uint8_t i;          // ringFrom peek index
889     uint16_t j;         // ASCII hex high nybble
890     uint16_t k;         // ASCII hex low nybble
```

```
    uint16_t ch;        // Character from ring, or EOF
    uint8_t sum = 0;    // Return value
    ringFrom = ring;
    for (i=0; i < 250; i++) {
        ch = peek(i);
        if (ch == EOF) break;  // There was no '*' on the end
        if ((0x20 < ch) && (ch < 0x7F)) sum += ch;      // Sum the byte
        if (ch == delimCS) {    // Has a NMEA '*' checksum delimiter
            ch = peek(i+1);
            if (ch == EOF) return sum + 0x0100;     // Low byte will be required checksum
            if (ch == delimCS) {// Allow '**' to be a valid checksum for testing
                if (peek(i+2) != delimCS) break;
                return sum; // Forced valid with **. This is what it should have been.
            }
            j = hexNybble(ch);
            if (j == EOF) break;   // Not an ASCII hex character
            k = hexNybble(peek(i+2));
            if (k == EOF) break;   // Not an ASCII hex character
            if (sum == (j * 16 + k)) {
                return sum;   // Valid sum found. Good.
            }
            break;   // Saw valid '*xx' but not valid
        }
    }
    ringFrom = saveFrom;
    return EOF;     // Has an * but is malformed
}
```

10.9.2.2 uint16_t putCtrlChar (uint16_t *ch*)

putCtrlChar Send the given character to the ringTo Ring Buffer as a printable form

Make the displayable part of a traditional ASCII character dump.

All Control characters (0x00 - 0x1F) are displayed with a caret: ^@ - ^_

Non-Printable characters 0x7F and above are converted to periods ('.').

Parameters

ch	Character to print

Returns

The value of the original character (not converted).

Definition at line 783 of file cdm.c.

```
{   // **putCtrlChar** Send the given character to the ringTo Ring Buffer as a printable form
    if (ch != EOF) {
        if (ch < 0x20) {
            ch += 0x40;   // Traditional ^@ - ^_ for non-printable characters
            put('^');
        }
        if (ch > 0x7E) ch = 0x2E;   // '.' for all high-order chars
        put(ch);
    }
    return ch;
}
```

10.9.2.3 uint16_t putDumpChar (uint16_t *ch*)

putDumpChar Send the given character to the ringTo Ring Buffer as a printable form

Make the displayable part of a traditional ASCII character dump.

Non-Printable characters are converted to periods ('.').

Parameters

ch	Character to print

Returns

The value of the original character (not converted).

Definition at line 773 of file cdm.c.

```
{   // **putDumpChar** Send the given character to the ringTo Ring Buffer as a printable form
    if (ch != EOF) {
        if (ch < 0x20) ch = 0x2E;   // Traditional '.' for non-printable characters
        if (ch > 0x7E) ch = 0x2E;
        put(ch);
    }
    return ch;
}
```

10.9.2.4 void traceDump (uint8_t *index*)

traceDump Send a copy of whatever is in ringFrom to the PCD-bus as a Znn: message

Parameters

index	

Definition at line 2411 of file cdm.c.

```
{   // **traceDump** Send a copy of whatever is in ringFrom to the PCD-bus as a Znn: message
    uint8_t saveTo = ringTo;
    uint8_t i;
    ringTo = ringCDtx;
    putS("S:"); putSix(eeGet32(nvSN));  // Make a message from me
    put(';');
    put('Z'); putSix(index); put(':');
    for (i=0; i<250; i++) if (putCtrlChar(peek(i)) == EOF) break;
//    peekCopy();
    cdBlindSend();
    ringTo = saveTo;
}
```

10.10 ModBus Functions

ModBus Support includes Master and Slave in both ASCII and RTU modes.

Data Structures

- struct ModbusBits_t

  ModbusBits_t *ModBus Configuration and Status Bits*

Enumerations

- enum modbusTimeoutSelector_t { t15, t35, turnaround, response }

  modbusTimeoutSelector_t *Different Timeout values used to implement ModBus*

Functions

- uint8_t parity (uint8_t ch)

  parity *Compute the parity of the character ch*
- uint16_t modbusCRC16 ()

  modbusCRC16 *Compute the Modbus CRC-16 for the contents of ringFrom*
- uint32_t modbusTimeout (modbusTimeoutSelector_t selector)

  modbusTimeout *Return the selected timeout value based on current baud rate*
- void Modbus (void)

  Modbus *Module is a ModBus Interface*

- ModbusBits_t ModbusBits

  Non-Volatile shadow at nvModbus (8 bits)

10.10.1 Detailed Description

ModBus Support includes Master and Slave in both ASCII and RTU modes.

10.10.2 Enumeration Type Documentation

10.10.2.1 enum modbusTimeoutSelector_t

modbusTimeoutSelector_t Different Timeout values used to implement ModBus

Different possible timeout values to be returned by modbusTimeout().

Enumerator

t15 1.5 character time timeout for message packets (inter-character)

t35 3.5 character time timeout for message packets (end of packet)

turnaround Turnaround Timeout.

response Response Timeout.

Definition at line 903 of file cdm.h.

```
{   // **modbusTimeoutSelector_t** Different Timeout values used to implement ModBus
    t15,
    t35,
    turnaround,
    response
} modbusTimeoutSelector_t;
```

10.10.3 Function Documentation

10.10.3.1 void Modbus (void)

Modbus Module is a ModBus Interface

Definition at line 2781 of file cdm.c.

```
{   // **Modbus** Module is a ModBus Interface
    uint32_t commTime;      // Time of last byte transferred
    uint8_t  prevBytes;     // Number of bytes in the ring
    uint16_t crc;
    sleepLevel(2);  // Aux Power On

    while (1) {
        if (ModbusBits.fMaster) {
            ringTo = ringSCR;
            ringFrom = ringSCR;
            put(0x01);      // SlaveID
            put(0x01);      // Modbus Function Code
                // ... Build the message into ringSCR
            crc = modbusCRC16();
            put (crc);
            put (crc>>8);
            ringCopy(ringTX, ringSCR);

            prevBytes = 0xFF;
            do {
                wait(0);
                if (prevBytes != txUsed) {
                    prevBytes = txUsed;
                    commTime = rtcMicroSeconds();
                }
            }
            while ((rtcMicroSeconds() - commTime) < modbusTimeout(
response));


        } else {    // It is a ModBus Slave

        }
    }


}
```

10.10.3.2 uint16_t modbusCRC16 ( )

modbusCRC16 Compute the Modbus CRC-16 for the contents of ringFrom

This can be used to compute the CRC-16 of a message received in ringRX, where we would expect the result to be 0x0000.

Or it can be used to look at ringSCR for a message to be sent. In this case the result would be appended to the message as the CRC using Little-Endian byte order.

CRC-16 computation compatible with ModBus. The initialization is 0xFFFF. The polynomial is 0xA001. The bit order is LSB first.

Returns

CRC-16 value for use with ModBus

Definition at line 919 of file cdm.c.

```
920 {   // **modbusCRC** Compute the Modbus CRC-16 for the contents of ringFrom
921     uint8_t i;              // ringFrom peek index
922     uint8_t j;              // Bit-per-byte loop counter
923     uint16_t ch;            // Character from ring, or EOF
924     uint16_t crc = 0xFFFF;  // Return value
925     for (i=0; ; i++) {
926         ch = peek(i);
927         if (ch == EOF) break;
928
929         crc ^= ch;      // XOR the next byte into the lower half of the crc
930         for (j=8; j; j--) { // For each bit
931             if (crc & 1) {  // test LSB
932                 crc >> 1;
933                 crc ^= 0xA001;  // Shift right and XOR polynomial value
934             } else {
935                 crc >> 1;   // Simply shift right
936             }
937         }
938     }
939     return crc;
940 }
```

10.10.3.3 uint32_t modbusTimeout (modbusTimeoutSelector_t *selector*)

modbusTimeout Return the selected timeout value based on current baud rate

Parameters

selector	

Returns

Number of microseconds for the selected timeout

Definition at line 942 of file cdm.c.

```
943 {   // **modbusTimeout** Return the selected timeout value based on current baud rate
944     uint32_t charTime = 500;   // microSeconds per char at greater than 19200 baud
945     switch (SerialBits.baud) {
946         case baud1200:  charTime = 8333;    break;
947         case baud2400:  charTime = 4167;    break;
948         case baud4800:  charTime = 2083;    break;
949         case baud9600:  charTime = 1042;    break;
950         case baud19200: charTime =  521;    break;
951     }
952     switch (selector) {
953         case t15:           return charTime   + charTime/2; // 1.5 * character time
954         case t35:           return charTime*3 + charTime/2; // 3.5 * character time
955         case turnaround:    return 100000;                  // 100mSec standard
956         case response:      return 1000000;                 // 1Sec standard
957     }
958     return 0;   // Something not selected correctly. Cause errors so we can fix it.
959 }
```

10.10.3.4 uint8_t parity (uint8_t *ch*)

parity Compute the parity of the character ch

Return 0 or 1 indicating the parity of character ch.

Parameters

ch	Character to get parity of

Returns

Parity of ch, either 0 or 1

Definition at line 877 of file cdm.c.

```
878 {   // **parity** Compute the parity of the character ch
879     ch ^= ch>>4;    // Hopefully the compiler improves this with a nybble swap
880     ch ^= ch>>2;
881     ch ^= ch>>1;
882     return (ch & 1);
883 }
```

10.10.4 Variable Documentation

10.10.4.1 ModbusBits_t ModbusBits

Non-Volatile shadow at nvModbus (8 bits)

ModBus configuration bits. Non-Volatile from nvModbus.

Definition at line 349 of file cdm.h.

10.11 Power-Clock-Data Bus Operation

The PCD-Bus provides for power supply and message exchange between modules.

Functions

- uint16_t crc8 (uint16_t iCRC, uint8_t b)

  crc8 *Compute updated value of CRC8 register for a given byte.*
- uint16_t cdGet (void)

  cdGet *Get the next character from the PCD-Bus Input message packet*
- uint16_t cdPeek (int16_t n)

  cdPeek *Peek at a character in the PCD-Bus Input message packet*
- uint16_t cdPut (uint16_t ch)

  cdPut *Add a character to the PCD-Bus Output message packet*
- void cdBlindSend (void)

  cdBlindSend *Send the contents of cdTxRing to the bus, discarding any Rx messages*
- void loopDelay (int16_t k)

  loopDelay *Insert a delay to compensate for hardware issued like cable length and capacitance*
- void cdPoll (void)

  cdPoll *Poll the PCD-Bus waiting for the start of a message packet*

Variables

- uint8_t cdDelayCount

  Loop Delays for CD-Bus Timeouts.

- enum cdState_t {
  cdWaitIdle, cdIdle, cdRecvData, cdTrmtData,
  cdRecvDone, cdTrmtDone }

  cdState_t *Allowed States for the PCD-Bus Interface*

10.11.1 Detailed Description

The PCD-Bus provides for power supply and message exchange between modules.

PCD-Bus Port pin usage:

```
C is portB0/latB0 with active pullup on trisB4

D is portB1/latB1 with active pullup on trisB5
```

The delay is for capacitive transition before the next possible op we ensure that the output latch bits are always set correctly This prevents the need for any explicit initialization of the port pins.

10.11.2 Enumeration Type Documentation

10.11.2.1 enum cdState_t

cdState_t Allowed States for the PCD-Bus Interface

Enumerator

cdWaitIdle Wait timeout in progress before becoming Idle.

cdIdle The PCD-Bus is idle waiting for the next message.

cdRecvData Message reception is in progress.

cdTrmtData Message transmission is in progress.

cdRecvDone A completed message is ready to be processed.

cdTrmtDone Transmit was successful.

Definition at line 294 of file cdm.h.

```
295 {   // Allowed States for the PCD-Bus Interface
296     cdWaitIdle,
297     cdIdle,
298     cdRecvData,
299     cdTrmtData,
300     cdRecvDone,
301     cdTrmtDone
302 } cdState_t;
```

10.11.3 Function Documentation

10.11.3.1 void cdBlindSend (void)

cdBlindSend Send the contents of cdTxRing to the bus, discarding any Rx messages

Returns

Definition at line 1407 of file cdm.c.

```
1408 {   // **cdBlindSend** Send the contents of cdTxRing to the bus, discarding any Rx messages
1409     while (cdTxUsed) {
1410         if (cdState == cdRecvDone) cdState = cdWaitIdle;
1411         if (cdState == cdTrmtDone) cdState = cdWaitIdle;
1412         wait(0);
1413     }
1414 }
```

10.11.3.2 uint16_t cdGet (void)

cdGet Get the next character from the PCD-Bus Input message packet

Returns

Definition at line 1369 of file cdm.c.

```
1370 {   // **cdGet** Get the next character from the PCD-Bus Input message packet
1371     if (cdRxUsed == 0) return EOF;
1372
1373     if (cdRxGetPtr >= sizeof cdRxRing) cdRxGetPtr = 0;
1374     cdRxUsed--;
1375     if (cdRxUsed == 0) cdState = cdWaitIdle; // Ring just became empty
1376     return cdRxRing[cdRxGetPtr++];
1377 }
```

10.11.3.3 uint16_t cdPeek (int16_t *n*)

cdPeek Peek at a character in the PCD-Bus Input message packet

Parameters

n	

Returns

Definition at line 1379 of file cdm.c.

```
1380 {   // **cdPeek** Peek at a character in the PCD-Bus Input message packet
1381     uint16_t i;
1382     if (cdRxUsed == 0) return EOF;
1383
1384     i = iPeekOffset(n, cdRxGetPtr, cdRxUsed, sizeof
      cdRxRing);
1385     if (i == EOF) return EOF;
1386     return cdRxRing[i];
1387 }
```

10.11.3.4 void cdPoll (void)

cdPoll Poll the PCD-Bus waiting for the start of a message packet

Handle the critical operation of transferring data over the PCD-bus. The state of the bus at any given time is cdState. If we are cdIdle we may begin receiving at any time. if we see clocks from another device on the bus.

If we discover characters in the cdTxRing we will begin transmission as soon as we are idle.

This means PCD-Bus output must be done in a monolithic manner between cdPolls. < Current Byte being received

< Current Byte being Transmitted

< Bit Number within Byte 8..1

< Receive Check Byte Being Built

< Transmit Check Byte Being Built

< Receive CRC Byte Being Built

< Transmit CRC Byte Being Built

Definition at line 1432 of file cdm.c.

```
{   // **cdPoll** Poll the PCD-Bus waiting for the start of a message packet
    uint8_t fSendChk;
    uint8_t cdRxByte;
    uint8_t cdTxByte;
    uint8_t cdBitCount;
    uint8_t cdRxChk;
    uint8_t cdTxChk;
    uint16_t cdRxCRC;
    uint16_t cdTxCRC;

    if (cdState == cdWaitIdle) {    // Release everything until C is high for timeout
        releaseC;   // Hands off while we reset
        releaseD;
        if (testC == 0) {
            if (cdDelayCount < 50) cdDelayCount = 0; else cdDelayCount = 100;
            // Restart delay if anybody is talking
            // Longer delay before we can try to transmit
        }
        if (cdDelayCount-- > 0) return;
        cdState = cdIdle;
        setDlow;
    }
    if (cdState == cdIdle) {    // will hold D low and release C
        if (testC == 0) cdState = cdRecvData; else
        if (cdTxUsed) cdState = cdTrmtData; else return;
        cdRxChk = 1;        // Fresh Check bytes
        cdTxChk = 1;
        cdRxCRC = 0;
        cdTxCRC = 0;
        cdBitCount = 8;     // First bit in byte
        cdTxGetPtr = 0;     // First Byte of message
        cdRxPutPtr = 0;
        cdRxGetPtr = 0;
        cdRxUsed = 0;
        cdRxByte = 0;       // Keep the compiler happy
    }
    if (cdState == cdTrmtData) {
        cdDiagTB++; //DIAG
        while (1) {
            if (cdBitCount == 8) {
                if (cdTxUsed > 0) {         // Get the next byte to send
                    if (cdTxGetPtr >= sizeof cdTxRing) cdTxGetPtr = 0;
                    cdTxByte = cdTxRing[cdTxGetPtr++];
                    cdTxChk += cdTxByte;
                    cdTxCRC = crc8(cdTxCRC, cdTxByte);
                    cdTxUsed--;
                    fSendChk = 1;
                } else
                if (fSendChk) {
                    cdTxByte = 0-cdTxChk;   // Get the check byte to send
                    cdTxByte = (uint8_t)(cdTxCRC >> 8);// Get the CRC to send
                    fSendChk = 0;
                } else {                    // Finish the transmission
                    releaseD;
                    cdDelayCount = 100;
                    while ((cdDelayCount--) && testC) continue;
                    if (testC == 0) break;   // Message continuation. Do receive
                    cdTxGetPtr = 0;
                    cdTxPutPtr = 0;
                    cdTxUsed = 0;
                    if (cdRxUsed) {cdRxUsed--; cdRxPutPtr--; }  // No check byte
                    cdState = cdTrmtDone;
                    setDlow;
                    return; // Transmitted message is complete
                }
                txPut(cdTxByte);
            }
            setClow;        // Begin sending a bit
            loopDelay(5);   // Clock goes low, then we set the new data bit
            releaseD;
            while (testD == 0) {
                while (testD == 0) {    // Wait for everyone else to release the bus
                    rxPoll();
                    txPoll();
                }
                loopDelay(5);   // Help ensure against noise on the D line
            }
            if ((cdTxByte & 0x80) == 0) setDlow; // This is the data
            loopDelay(5);
```

```
            setChigh;
            loopDelay(5);
            while (testC == 0) continue;    // End of clock extension
            if (cdTxByte & 0x80) {  // We want to send a 1
                if (testD == 0) break;  // Somebody else is asserting other data
            }
            cdTxByte <<= 1;
            loopDelay(50);  // This is the transmit bit rate
            cdRxByte <<= 1;
            if (testD) cdRxByte++;  // Receive the bit we are sending
            if (cdBitCount-- == 1) {
                // If we have another byte to send, we must grab the clock now.
                // The Bytewise CRC takes a long time and we have to do RX and TX
                if (fSendChk) setClow;  // Prevent inter-character timeouts
                cdRxCRC = crc8(cdRxCRC, cdRxByte);
                cdRxChk += cdRxByte;
                if (cdRxPutPtr >= sizeof cdRxRing) cdRxPutPtr = 0;
                cdRxRing[cdRxPutPtr++] = cdRxByte;
                cdRxUsed++;
                cdBitCount = 8;
            }
        }
        // Somebody else's data. Switch to receiving.
        cdDiagTE++; //DIAG
        cdTxGetPtr = 0;         // Restore the cdTxRing so we can try again
        cdTxUsed = cdTxPutPtr;
        cdState = cdRecvData;   // Switch to receiving to get the message
    }
    if (cdState == cdRecvData) {
        setDhigh;
        cdDiagRB++; //DIAG
        while (1) {
            if (testC) {
                cdRxByte <<= 1;
                if (testD) cdRxByte++; // Receive the bit we are sending
                if (cdBitCount-- == 1) {
                    cdRxCRC = crc8(cdRxCRC, cdRxByte);
                    cdRxChk += cdRxByte;
                    if (cdRxPutPtr >= sizeof cdRxRing) cdRxPutPtr = 0;
                    cdRxRing[cdRxPutPtr++] = cdRxByte;
                    cdRxUsed++;
                    cdDelayCount = 100;
                    while ((cdDelayCount--) && testC) continue;
                    if (testC) {     // message received OK
                        setDlow;
                        // Ignore Message if Check Byte Error
                        if ((uint8_t)(cdRxCRC >> 8)) {
                            cdDiagRD++; //DIAG
                            cdState = cdIdle;
                        } else {
                            cdRxPutPtr--;   // Check Byte disappears
                            cdRxUsed--;
                            cdState = cdRecvDone;
                        }
                        return;
                    }
                    cdBitCount = 8;
                }
                cdDelayCount = 100;
                while ((cdDelayCount--) && testC) continue;
                if (testC) {     // framing error. Clock stayed high too long
                    cdDiagRE++; //DIAG
                    cdState = cdWaitIdle;
                    return;
                }
                setDlow;
                txPoll();
                rxPoll();
                setDhigh;
            }
        }
    }
}
```

10.11.3.5 uint16_t cdPut (uint16_t *ch*)

cdPut Add a character to the PCD-Bus Output message packet

Parameters

ch	

Returns

Definition at line 1389 of file cdm.c.

```
1390 {   // **cdPut** Add a character to the PCD-Bus Output message packet
1391     if (ch != EOF) {
1392         if (cdTxUsed >= sizeof cdTxRing) {
1393             // Check for output ring overrun and discard the character
1394             ErrorBits.fcdTxRingOver = 1;
1395             return EOF;
1396         }
1397         // Reset the cdTxRing if it is empty
1398         if (cdTxUsed == 0) {cdTxPutPtr = 0; cdTxGetPtr = 0; }
1399         // Wrap the ring if needed
1400         if (cdTxPutPtr >= sizeof cdTxRing) cdTxPutPtr = 0;
1401         cdTxRing[cdTxPutPtr++] = ch;
1402         cdTxUsed++;
1403     }
1404     return ch;
1405 }
```

10.11.3.6 uint16_t crc8 (uint16_t *iCRC*, uint8_t *b*)

crc8 Compute updated value of CRC8 register for a given byte.

This is the 8-bit CRC used by the PCD-Bus messages.

Initialize the iCRC value to zero; expect (uint8_t)(iCRC>>8) zero when successful.

Parameters

iCRC	Running CRC word. Initialize to zero
b	Next byte to be added into the CRC

Returns

New running CRC word.

Definition at line 1416 of file cdm.c.

```
1417 {   // **crc8** Compute updated value of CRC8 register for a given byte.
1418     uint8_t i;
1419     iCRC ^= b << 8;
1420     for (i=8;i;i--) {
1421         if (iCRC & 0x8000) iCRC^= (0x1070 << 3);
1422         iCRC <<= 1;
1423     }
1424     return iCRC;
1425 }
```

10.11.3.7 void loopDelay (int16_t *k*)

loopDelay Insert a delay to compensate for hardware issued like cable length and capacitance

Definition at line 1427 of file cdm.c.

```
1428 {   // **loopDelay** Insert a delay to compensate for hardware issues like cable length and capacitance
1429     while(k--) continue;
1430 }
```

10.11.4 Variable Documentation

10.11.4.1 uint8_t cdDelayCount

Loop Delays for CD-Bus Timeouts.

Definition at line 1328 of file cdm.h.

10.12 Peripheral Device Polling

Polling of hardware peripherals that are not handled by interrupts.

Functions

- void pollCritical (void)

  pollCritical *Poll the time-critical hardware*
- uint8_t wait (uint8_t secs)

  wait *Wait the specified number of seconds while ensuring that all polling is handled*

Variables

- uint16_t pollTime

  Timer value at previous Critical Poll.

10.12.1 Detailed Description

Polling of hardware peripherals that are not handled by interrupts.

10.12.2 Function Documentation

10.12.2.1 void pollCritical (void)

pollCritical Poll the time-critical hardware

Definition at line 1313 of file cdm.c.

```
1314 {   // **pollCritical** Poll the time-critical hardware
1315         // Measure the interval between critical polls for diagnostic purposes.
1316         // We must ensure that we do not allow any overruns since
1317         // these are not interrupt-driven.
1318     int16_t now = TMR0;
1319     int16_t interval = now - pollTime;  // Number of clocks since last poll
1320     if (interval < 0) interval += rtcReload;    // THIS IS WRONG!!!
1321     if (interval > pollLongest) pollLongest = interval;
1322     pollTime = now;
1323
1324         // Handle the actual polling of autonomous hardware devices here
1325     cdPoll();   // Make sure we handle the Clock and Data bus bits
1326     rxPoll();   // Buffer received characters while waiting
1327     txPoll();   // Send Serial characters while waiting
1328
1329     if (ModeBits.fPollLED) {   // Handle non-ISR LED blinking
1330         if (ModeBits.fDiagLED == 0) {
1331             if (ledMask == 0) ledMask = 0x80;
1332             leftLED = (ledA & ledMask) ? 1 : 0;    // Blink the LEDs
1333             rightLED = (ledB & ledMask) ? 1 : 0;
1334             ledMask >>= 1;
1335             if (ledMask == 0) {
1336                 ledMask = 0x80;
1337                 ledA = ledAreload;  // LED blinking is always synchronized
1338                 ledB = ledBreload;  // and uses all 8 bits of the pattern
1339             }
1340         }
1341     }
1342 }
```

10.12.2.2 uint8_t wait (uint8_t *secs*)

wait Wait the specified number of seconds while ensuring that all polling is handled

Wait the given number of seconds and poll all hardware functions. Use 0 seconds to ensure that polling is handled while doing other operations. The return value 1 indicates that we saw a one-second tick.

Parameters

secs	Seconds to wait

Returns

Return 0 means we did not see a tick, 1 indicates a tick

Definition at line 1344 of file cdm.c.

```
1345 {   // **wait** Wait the specified number of seconds while ensuring that all polling is handled
1346     uint8_t saveRingTo;
1347     while (1) {
1348         pollCritical();
1349         diPoll();
1350         uiPoll();
1351         adcPoll();
1352         if (rtcPoll()) {
1353             if (ModeBits.fUI && ModeBits.fUIclock) {
1354                 saveRingTo = ringTo;
1355                 ringTo = ringDisp; diCursor = 0;
1356                 rtcDateTime (0x65);     // 3d12:34:56 elapsed time since reset
1357                 diBlankField(15);
1358                 ringTo = saveRingTo;
1359             }
1360             if (secs == 0) break;   // zero timeout but we saw a Tick
1361             secs--;
1362             if (secs == 0) break;   // Final timeout with a Tick
1363         }
1364         if (secs == 0) return 0;    // zero timeout and we did not Tick
1365     }
1366     return 1;   // a Tick occurred during this wait
1367 }
```

10.12.3 Variable Documentation

10.12.3.1 uint16_t pollTime

Timer value at previous Critical Poll.

Definition at line 1248 of file cdm.h.

10.13 Power Management

Handling of peripheral power and Sleep Mode.

Macros

- #define fPower (LATAbits.LA3)

  Output bit that controls peripheral power.

Functions

- void sleepLevel (uint8_t level)

  sleepLevel *Select Power level: 0=sleep, 1=running, 2=Aux On*

10.13.1 Detailed Description

Handling of peripheral power and Sleep Mode.

10.13.2 Macro Definition Documentation

10.13.2.1 #define fPower (LATAbits.LA3)

Output bit that controls peripheral power.

Definition at line 1238 of file cdm.h.

10.13.3 Function Documentation

10.13.3.1 void sleepLevel (uint8_t *level*)

sleepLevel Select Power level: 0=sleep, 1=running, 2=Aux On

Definition at line 817 of file cdm.c.

```
818 {   // **sleepLevel** Select Power level: 0=sleep, 1=running, 2=Aux On
819     switch (level) {
820         case 0:
821             adcDisable();           // Analog Channels and voltage reference disabled
822
823             leftLEDtris = 1;        // LEDs off by making them inputs
824             rightLEDtris = 1;
825
826             fPower = 0;              // Aux Power Supply disabled
827
828             RCSTA1bits.SPEN = 0;    // Disable the serial port
829             RTSout = 0;
830
831             // Slow the system clock here
832
833                 // Here we are in low-power sleep mode
834             setDlow;    // Indicate that we are busy sleeping
835             releaseC;   // Make sure we are not holding the bus
836             while (testC) {     // PCD-Bus Clock idles high during sleep
837                 rtcPoll();
838                 if (rtcTimerA == 0) break;  // Wake up if any of the timers have expired
839                 if (rtcTimerB == 0) break;
840                 if (rtcTimerC == 0) break;
```

```

                ;  // Here we actually sleep. The timer will wake us

                // Compensate for slow ticks for the next poll here
            }
            // We woke up so we fall through to restore normal operation
        case 1:
            // Restore the system clock here

                // Wake up and restore everything important
            ANSELA = 0x07; // Disable the Analog Inputs of PORTA Leave 3 Analog Inputs
            ANSELB = 0; // Disable the Analog Inputs of PORTB
            ANSELC = 0; // Disable the Analog Inputs of PORTC
            LATA = 0;
            TRISA = 0x07; // PORTA outputs for LEDs and PSon
            TRISB = 0xFF; // PORTB is all inputs while idle. No effect on CD-Bus
            INTCON2bits.RBPU = 0; // Allow Weak Pull-ups on PortB
            WPUB = 0x03; // Weak Pullups only on the C and D lines
            LATC = 0x80;
            TRISC = 0x80; // PORTC is driven low except for the USART RX line

            RCSTA1bits.SPEN = 0;    // Disable the serial port
            leftLEDtris = 0;    // LEDs enabled again
            rightLEDtris = 0;
            fPower = 0;
            break;
        case 2:
            if (fPower == 0) {
                fPower = 1;
                ledMask = 0; ledA = 0xFF; ledB = 0xFF;  // Double Long LED
                wait(2);    // Time for the power to stabilize - crude
            }
            RCSTA1bits.SPEN = 1;    // Enable the serial port
    }
}
```

10.14 Ring Buffers

Ring Buffers allow consistent access to strings of characters in different types of memory or devices or messages.

Enumerations

- enum ringSelect_t {
  ringNULL, ringRX, ringTX, ringCDrx,
  ringCDtx, ringSCR, ringRES, ringMSG,
  ringUSB, ringEE, ringCMP, ringDisp }

  ringSelect_t *Selection of the ring Buffers for get(), peek() and put()*

Functions

- uint16_t pos (const uint8_t *str)

  pos *Scan the selected ringFrom for the string, return the position*
- uint8_t scan (const uint8_t *str)

  scan *Flush the selected ringFrom up through the string*
- uint8_t scanCopy (const uint8_t *str)

  scanCopy *Copy ringFrom to ringTo up to but not including string*
- uint16_t put (uint16_t ch)

  put *Put the character ch into the Ring ringTo and handle leading zeroes and CR/LF*
- uint16_t get (void)

  get *Read and remove the next character from ringFrom (if any).*
- void ringReset (ringSelect_t ring)

  ringReset *Reset the pointers for the specified ring buffer, leaving it empty*
- uint16_t peek (int16_t n)

  peek *Peek at character n in ringFrom counting from ring beginning or end*
- uint16_t iPeekOffset (int16_t n, uint8_t getPtr, uint8_t used, uint8_t size)

  peekOffset *Compute the offset of an indexed charactter from beginning or end of a ring*
- void ringCopy (ringSelect_t ringT, ringSelect_t ringF)

  ringCopy *Copy the contents ringFrom to ringTo*
- void peekCopy (ringSelect_t ringT, ringSelect_t ringF)

  peekCopy *Copy the contents of ringFrom to ringTo without modifying ringFrom*
- uint16_t scrGet (void)

  scrGet *Read and remove the next character from scrRing (if any)*
- uint16_t scrPeek (int16_t n)

  scrPeek *Read character n counting from the beginning or end of scrRing*
- uint16_t scrPut (uint16_t ch)

  scrPut *Add the character ch to the end of scrRing*
- uint16_t msgGet (void)

  msgGet *Read and remove the next character from msgRing (if any)*
- uint16_t msgPeek (int16_t n)

  msgPeek *Read character n counting from the beginning or end of msgRing*
- uint16_t msgPut (uint16_t ch)

  msgPut *Add the character ch to the end of msgRing*
- uint16_t resGet (void)

resGet *Read and remove the next character from resRing (if any)*

- uint16_t resPeek (int16_t n)

  resPeek *Read character n counting from the beginning or end of resRing*

- uint16_t resPut (uint16_t ch)

  resPut *Add the character ch to the end of resRing*

- uint16_t cmpPut (uint16_t ch)

  cmpPut *Compare ch with the next character in ringFrom and modify fMismatch*

- #define EOF (0xFFFF)

  End of File signal for buffered reception.

- #define cdRingSize (150)

  Number of bytes allowed in cdRxRing and cdTxRing.

- #define rxRingSize (150)

  Number of bytes allowed in rxRing.

- #define txRingSize (150)

  Number of bytes allowed in txRing.

- #define msgRingSize (150)

  Number of bytes allowed in msgRing.

- #define scrRingSize (150)

  Number of bytes allowed in scrRing.

- #define resRingSize (150)

  Number of bytes allowed in resRing.

- uint8_t cdTxRing [cdRingSize]

  Ring Buffer for bytes to be transmitted over the PCD-Bus.

- uint8_t cdRxRing [cdRingSize]

  Ring Buffer for bytes being received over the PCD-Bus.

- uint8_t rxRing [rxRingSize]

  UART Receive Ring buffer.

- uint8_t txRing [txRingSize]

  Ring Buffer data automatically sent to the UART.

- uint8_t msgRing [msgRingSize]

  Message Ring buffer.

- uint8_t scrRing [scrRingSize]

  Scratch Ring buffer.

- uint8_t resRing [resRingSize]

  Response Ring buffer.

- ringSelect_t ringFrom

  Source for get() and peek()

- ringSelect_t ringTo

  Destination for put(ch)

- uint8_t cdRxGetPtr

  Index of next character to be read from the PCD-Bus Receive Ring.

- uint8_t cdRxPutPtr

  Index of next character to be placed in the PCD-Bus Receive Ring.

- uint8_t cdRxUsed

  Bytes used in cdRxRing.

- uint8_t cdTxGetPtr

Index of next character to be sent from the PCD-Bus Transmit Ring.

- uint8_t cdTxPutPtr

  Index of next character to be placed in the PCD-Bus Transmit Ring.
- uint8_t cdTxUsed

  Bytes used in cdTxRing.
- uint8_t rxGetPtr

  index to get next byte from rxRing
- uint8_t rxPutPtr

  Index to put next byte into rxRing.
- uint8_t rxUsed

  Bytes used in rxRing.
- uint8_t rxEcho

  Character to echo to the UART Transmit hardware.
- uint8_t txGetPtr

  Index of next character to send in the UART Transmit Ring.
- uint8_t txPutPtr

  Index of next character to append to the UART Transmit Ring.
- uint8_t txUsed

  Bytes used in txRing.
- uint8_t msgGetPtr

  index to get next byte from msgRing
- uint8_t msgPutPtr

  Index to put next byte into msgRing.
- uint8_t msgUsed

  Bytes used in msgRing.
- uint8_t scrGetPtr

  index to get next byte from scrRing
- uint8_t scrPutPtr

  Index to put next byte into scrRing.
- uint8_t scrUsed

  Bytes used in scrRing.
- uint8_t resGetPtr

  index to get next byte from resRing
- uint8_t resPutPtr

  Index to put next byte into resRing.
- uint8_t resUsed

  Bytes used in resRing.

10.14.1 Detailed Description

Ring Buffers allow consistent access to strings of characters in different types of memory or devices or messages.

The Ring Buffer mechanism is used to allow access to data messages, peripheral devices and different types of memory using only three functions:

```
peek(n)

get()

put(ch)
```

10.14.2 Macro Definition Documentation

10.14.2.1 #define cdRingSize (150)

Number of bytes allowed in cdRxRing and cdTxRing.

Definition at line 364 of file cdm.h.

10.14.2.2 #define EOF (0xFFFF)

End of File signal for buffered reception.

Definition at line 362 of file cdm.h.

10.14.2.3 #define msgRingSize (150)

Number of bytes allowed in msgRing.

Definition at line 374 of file cdm.h.

10.14.2.4 #define resRingSize (150)

Number of bytes allowed in resRing.

Definition at line 380 of file cdm.h.

10.14.2.5 #define rxRingSize (150)

Number of bytes allowed in rxRing.

Definition at line 368 of file cdm.h.

10.14.2.6 #define scrRingSize (150)

Number of bytes allowed in scrRing.

Definition at line 377 of file cdm.h.

10.14.2.7 #define txRingSize (150)

Number of bytes allowed in txRing.

Definition at line 371 of file cdm.h.

10.14.3 Enumeration Type Documentation

10.14.3.1 enum ringSelect_t

ringSelect_t Selection of the ring Buffers for get(), peek() and put()

All possible real or virtual Ring Buffers.

Enumerator

ringNULL Bit bucket.

ringRX Receive Ring Buffer to get() for the serial port.

ringTX Transmit Ring Buffer to put() for the serial port.

ringCDrx Receive Ring Buffer to get() for the PCD-Bus.

ringCDtx Transmit Ring Buffer to put() for the PCD-Bus.

ringSCR Selects get() or put() for the Scratch Ring Buffer.

ringRES Selects get() or put() for the Response Ring Buffer.

ringMSG Selects get() or put() Buffers for the Message Ring Buffer.

ringUSB Selects get() or put() Buffers for the USB port implementation.

ringEE Virtual get() or put() Ring that allows reading from and writing to EEPROM.

ringCMP Virtual put() Ring for character by character comparison with the Scratch Ring Buffer.

ringDisp Output Buffer for put() to the 2x16 character LCD Display.

Definition at line 210 of file cdm.h.

```
{   // **ringSelect_t** Selection of the ring Buffers for get(), peek() and put().
    ringNULL,
    ringRX,
    ringTX,
    ringCDrx,
    ringCDtx,
    //ringCD,     ///< Selects get() or put() Ring Buffer for the PCD-Bus
    ringSCR,
    ringRES,
    ringMSG,
    ringUSB,
    ringEE,
    ringCMP,
    ringDisp
} ringSelect_t;
```

10.14.4 Function Documentation

10.14.4.1 uint16_t cmpPut (uint16_t *ch*)

cmpPut Compare ch with the next character in ringFrom and modify fMismatch

This is a virtual ring that compares the characters we put() here against the characters in ringFrom.

If there is a mismatch we set CompareBits.fMismatch.

Begin by setting

```
CompareBits.fMismatch = 0;  // Clear previous comparison
CompareBits.fAlpha = 1;     // Ignore punctuation
CompareBits.fWild = 1;      // Allow '?' to match any single character
ringTo = ringCMP;
ringFrom = ...;  // Whichever ring is to be compared against
```

... Send characters as needed using put(ch)

When we are done sending, the program should detect strings not equal as follows

```
if (CompareBits.fMismatch || (peek(0) != EOF)) //strings not equal
```

Parameters

ch	

Returns

the same input character ch

Definition at line 333 of file cdm.c.

```
334 {   // **cmpPut** Compare ch with the next character in ringFrom and modify fMismatch
335     uint16_t cx;
336     if (CompareBits.fAlphaOnly) {
337
339
340         // Discard non-alpha characters as if they were not there
341         if (!isAlpha(ch)) return ch;
342         while (1) {
343             cx = peek(0);
344             if (cx == EOF) {
345                 if (ch != EOF) CompareBits.fMismatch = 1;
346                 return ch;
347             }
348             if (isAlpha(cx)) break;
349             get();   // Discard the non-alpha character from ringSCR
350         }
351     }
352             // '?' is single-character wildcard. Any real character (not EOF) is OK.
353     if (CompareBits.fWild && (ch != '?')) {
354         if (get() == EOF) CompareBits.fMismatch = 1;
355     } else {
356         if (get() != ch) CompareBits.fMismatch = 1;
357     }
358     return ch;
359 }
```

10.14.4.2 uint16_t get (void)

get Read and remove the next character from ringFrom (if any).

Global variable ringFrom selects the (real or virtual) Ring Buffer to read from. The next available character is removed from the Ring and returned.

Returns

Return the next byte from the Ring Buffer or EOF if none available.

Definition at line 96 of file cdm.c.

```
97 {   // **get** Read and remove the next character from ringFrom (if any).
98     uint8_t ch;
99     switch (ringFrom) {
100         case ringCDrx:  return cdGet();
101         case ringEE:    ch = eeGet(EEADR); if (ch==0) return EOF; EEADR++; return ch;
102         case ringSCR:   return scrGet();
103         case ringRES:   return resGet();
104         case ringRX:    return rxGet();
105         case ringMSG:   return msgGet();
106 //        case ringUSB:   return usbGet();
107     }
108     return EOF;
109 }
```

10.14.4.3 uint16_t iPeekOffset (int16_t *n*, uint8_t *getPtr*, uint8_t *used*, uint8_t *size*)

peekOffset Compute the offset of an indexed charactter from beginning or end of a ring

Given the getPtr, bytes used and size of a Ring Buffer, compute the actual offset of character n in the Ring Buffer.

Positive values of n count forward from the oldest character. Negative values of n count backward from the most recent character.

Parameters

n	index of the character within the active part of the Ring Buffer
getPtr	Current getPtr of the Ring Buffer
used	Current number of characters Used in the Ring Buffer
size	Size of the Ring Buffer

Returns

Actual index of character n within the buffer, or EOF

Definition at line 67 of file cdm.c.

```
68 {   // **peekOffset** Compute the offset of an indexed character from beginning or end of a ring
69     if (used == 0) return EOF;
70     if (n >= 0) {
71         if (n >= used) return EOF;
72     } else {
73         n += used;
74         if (n < 0) return EOF;
75     }
76     n += getPtr;                // Make actual ring offsetfrom getPtr
77     if (n >= size) n -= size;   // Wrap the ring if needed
78
79     return n;
80 }
```

10.14.4.4 uint16_t msgGet (void)

msgGet Read and remove the next character from msgRing (if any)

Returns

Definition at line 259 of file cdm.c.

```
260 {   // **msgGet** Read and remove the next character from msgRing (if any)
261     if (msgUsed == 0) return EOF; // Ring is empty
262     msgUsed--;
263     if (msgGetPtr >= sizeof msgRing) msgGetPtr = 0;
264     return msgRing[msgGetPtr++];
265 }
```

10.14.4.5 uint16_t msgPeek (int16_t *n*)

msgPeek Read character n counting from the beginning or end of msgRing

Parameters

n	

Returns

Definition at line 267 of file cdm.c.

```
268 {   // **msgPeek** Read character n counting from the beginning or end of msgRing
269     uint16_t i;
270     if (msgUsed == 0) return EOF; // Ring is empty
271     i = iPeekOffset(n, msgGetPtr, msgUsed, sizeof
      msgRing);
272     if (i == EOF) return EOF;   // peek past beginning or end of actual data
273     return msgRing[i];
274 }
```

10.14.4.6 uint16_t msgPut (uint16_t *ch*)

msgPut Add the character ch to the end of msgRing

Parameters

ch	

Returns

the same input character ch

Definition at line 276 of file cdm.c.

```
277 {   // **msgPut** Add the character ch to the end of msgRing
278     if (ch == EOF) return EOF;
279     if (msgUsed == sizeof msgRing) {
280         ErrorBits.fmsgRingOver = 1; // msgRing is full
281         return EOF;
282     } else {
283         if (msgUsed == 0) {msgPutPtr = 0; msgGetPtr = 0; }   // Reset msgRing
284         if (msgPutPtr >= sizeof msgRing) msgPutPtr = 0; // wrap putPtr to the
      beginning
285         msgRing[msgPutPtr++] = ch;
286         msgUsed++;
287     }
288     return ch;
289 }
```

10.14.4.7 uint16_t peek (int16_t *n*)

peek Peek at character n in ringFrom counting from ring beginning or end

Global variable ringFrom selects the (real or virtual) Ring Buffer to peek at. The contents of the Ring Buffer are not modified. i.e., the character is NOT removed.

Parameters

n	Offset of character within the Ring Buffer to return. 0 indicates the oldest character (the one that the next get() will return). positive numbers indicate subsequent characters. negative numbers index backwards from the newest character added.

Returns

Return the indexed byte from the Ring Buffer or EOF if not available.

Definition at line 82 of file cdm.c.

```
{   // **peek** Peek at character n in ringFrom counting from ring beginning or end
    uint8_t ch;
    switch (ringFrom) {
        case ringCDrx:  return cdPeek(n);
        case ringEE:    ch = eePeek(n); if (ch==0) return EOF; return ch;
        case ringSCR:   return scrPeek(n);
        case ringRES:   return resPeek(n);
        case ringRX:    return rxPeek(n);
//        case ringUSB:   return usbPeek(n);
    }
    return EOF;
}
```

10.14.4.8 void peekCopy (ringSelect_t *ringT*, ringSelect_t *ringF*)

peekCopy Copy the contents of ringFrom to ringTo without modifying ringFrom

The contents of the Ring Buffer given by ringFrom will be copied to the Ring Buffer indicated by ringTo.

Characters are NOT removed from the ringFrom buffer

If the destination Ring Buffer overflows we stop. Thus, it is possible to get an incomplete copy.

Parameters

ringT	
ringF	

Definition at line 161 of file cdm.c.

```
{   // **peekCopy** Copy the contents of ringFrom to ringTo without modifying ringFrom
    ringSelect_t saveTo = ringTo;
    ringSelect_t saveFrom = ringFrom;
    ringTo = ringT;
    ringFrom = ringF;
    uint8_t i;
    for(i=0; put(peek(i))!=EOF; i++);   // Peek until EOF
    ringTo = saveTo;
    ringFrom = saveFrom;
}
```

10.14.4.9 uint16_t pos (const uint8_t * *str*)

pos Scan the selected ringFrom for the string, return the position

Scan the selected ringFrom for string. If found, return the offset (0..n) If not found return EOF.

This never modifies the ring.

Parameters

str	string to search for

Returns

index if the chars found, EOF if not.

Definition at line 9 of file cdm.c.

```
{   // **pos** Scan the selected ringFrom for the string, return the position
    uint8_t i;
    uint8_t j;
    if (str[0] == 0) return 0;   // Make sure we handle null strings correctly
    for (i=0; i < 250; i++) {
        if (peek(i) == str[0]) {
            for (j=1; j < 250; j++) {
                if (str[j] == 0) return i;  // This is successful match
                if (peek(i+j) != str[j]) break;
            }
        }
    }
    return EOF; // No Match Found
}
```

10.14.4.10 uint16_t put (uint16_t *ch*)

put Put the character ch into the Ring ringTo and handle leading zeroes and CR/LF

Global variable ringTo selects the (real or virtual) Ring Buffer to append to. If the EOF value is passed, the contents of the Ring Buffer are not modified.

put(ch) implements the Zero Suppression feature. If FlagBits.fZeroSupp is set and a zero is to be sent, the character is suppressed and put(ch) returns EOF. Any non-'0' character is appended normally and the FlagBits.fZeroSupp is cleared.

put(ch) implements the Multiple CR/LF Suppression feature. Any character sent here sets FlagBits.fAllowCRLF.

Parameters

ch	Character to append to the selected Ring Buffer. EOF value indicates character is to be ignored (Buffer remains unchanged).

Returns

Return the value of ch, or EOF if the Ring would have overflowed or if the character was zero-suppressed.

Definition at line 111 of file cdm.c.

```
{   // **put** Put the character ch into the Ring ringTo and handle leading zeroes and CR/LF
    if (FlagBits.fZeroSupp) {
        if (ch == '0') return EOF;  // Suppress the leading zeroes
        FlagBits.fZeroSupp = 0;     // Stop suppression at any non-zero character
    }
    if (ch == EOF) return EOF;
    FlagBits.fAllowCRLF = 1;    // Any character actually output here makes CR/LF ok
    switch (ringTo) {
        case ringCDtx:  return cdPut(ch);
        case ringDisp:  return diPut(ch);
        case ringEE:    return eePut(ch);
        case ringMSG:   return msgPut(ch);
        case ringSCR:   return scrPut(ch);
        case ringRES:   return resPut(ch);
        case ringTX:    return txPut(ch);
        case ringCMP:   return cmpPut(ch);
//        case ringUSB:   return usbPut(ch);
    }
    return EOF;
}
```

10.14.4.11 uint16_t resGet (void)

resGet Read and remove the next character from resRing (if any)

Returns

Definition at line 291 of file cdm.c.

```
292 {   // **resGet** Read and remove the next character from resRing (if any)
293     if (resUsed == 0) return EOF; // Ring is empty
294     resUsed--;
295     if (resGetPtr >= sizeof resRing) resGetPtr = 0;
296     return resRing[resGetPtr++];
297 }
```

10.14.4.12 uint16_t resPeek (int16_t *n*)

resPeek Read character n counting from the beginning or end of resRing

Parameters

n	

Returns

Definition at line 299 of file cdm.c.

```
300 {   // **resPeek** Read character n counting from the beginning or end of resRing
301     uint16_t i;
302     if (resUsed == 0) return EOF; // Ring is empty
303     i = iPeekOffset(n, resGetPtr, resUsed, sizeof
      resRing);
304     if (i == EOF) return EOF;   // peek past beginning or end of actual data
305     return resRing[i];
306 }
```

10.14.4.13 uint16_t resPut (uint16_t *ch*)

resPut Add the character ch to the end of resRing

Parameters

ch	

Returns

the same input character ch

Definition at line 308 of file cdm.c.

```
309 {   // **resPut** Add the character ch to the end of resRing
310     if (ch == EOF) return EOF;
311     if (resUsed == sizeof resRing) {
312         ErrorBits.fresRingOver = 1; // resRing is full
```

```
        return EOF;
    } else {
        if (resUsed == 0) {resPutPtr = 0; resGetPtr = 0; }   // Reset resRing
        if (resPutPtr >= sizeof resRing) resPutPtr = 0; // wrap putPtr to the
beginning
        resRing[resPutPtr++] = ch;
        resUsed++;
    }
    return ch;
}
```

10.14.4.14 void ringCopy (ringSelect_t *ringT*, ringSelect_t *ringF*)

ringCopy Copy the contents ringFrom to ringTo

The contents of the Ring Buffer given by ringFrom will be copied to the Ring Buffer indicated by ringTo.

If the destination Ring Buffer overflows we stop. Thus, it is possible to still have characters remaining in the source after we are done.

Parameters

ringT	
ringF	

Definition at line 149 of file cdm.c.

```
{   // **ringCopy** Copy the contents of ringFrom to ringTo
    ringSelect_t saveTo = ringTo;
    ringSelect_t saveFrom = ringFrom;
    ringTo = ringT;
    ringFrom = ringF;
    while (put(get()) != EOF) continue; // Copy until EOF
    ringTo = saveTo;
    ringFrom = saveFrom;
}
```

10.14.4.15 void ringReset (ringSelect_t *ring*)

ringReset Reset the pointers for the specified ring buffer, leaving it empty

Reset the pointers for the selected ring buffer to make it empty.

For ringCDrx, if the ring was ready and not empty we also set the cdState to cdWaitIdle.

Parameters

ring	index of the actual ring whose pointers we will reset.

Definition at line 133 of file cdm.c.

```
{   // **ringReset** Reset the pointers for the specified ring buffer, leaving it empty
    switch (ring) {
        case ringCDtx:  cdTxGetPtr = 0; cdTxPutPtr = 0;
cdTxUsed = 0;   return;
        case ringSCR:   scrGetPtr = 0;  scrPutPtr = 0;
scrUsed = 0;    return;
        case ringRES:   resGetPtr = 0;  resPutPtr = 0;
resUsed = 0;    return;
        case ringRX:    rxGetPtr = 0;   rxPutPtr = 0;
rxUsed = 0;     return;
        case ringTX:    txGetPtr = 0;   txPutPtr = 0;
txUsed = 0;     return;
        case ringCDrx:
            // Ensure this behaves like the final get() from the ring and re-enables the CD-Bus
            if (cdState == cdRecvDone) cdState =
```

```
    cdWaitIdle;
144             if (cdState == cdTrmtDone) cdState =
    cdWaitIdle;
145             cdRxGetPtr = 0; cdRxPutPtr = 0; cdRxUsed = 0;   return;
146     }
147 }
```

10.14.4.16 uint8_t scan (const uint8_t * *str*)

scan Flush the selected ringFrom up through the string

Scan the selected ringFrom for string. If found, flush the ring up to and including the string and return 1 If not found do not modify the ring and return 0.

Parameters

str	string to search for

Returns

1 if the chars found, 0 if not.

Definition at line 25 of file cdm.c.

```
26 {   // **scan** Flush the selected ringFrom up through the string
27     uint16_t ch;
28     uint8_t j;
29     if (str[0] == 0) return 1;   // Null string is always found
30     while ((ch = peek(0)) != EOF) {
31         if (ch == str[0]) {
32             for (j=1; j < 250; j++) {
33                 if (str[j] == 0) {
34                     // Now discard the matching string
35                     for (; j>0; j--) get();
36                     return 1;  // This is successful match
37                 }
38                 if (peek(j) != str[j]) break;
39             }
40         }
41         get(); // Discard the ringFrom character
42     }
43     return 0; // String not found
44 }
```

10.14.4.17 uint8_t scanCopy (const uint8_t * *str*)

scanCopy Copy ringFrom to ringTo up to but not including string

Copy characters from ringFrom to ringTo until the string is found. If found, discard the matching string and return 1 If not found ringFrom will be empty and return 0.

Parameters

str	string to search for

Returns

1 if the chars found, 0 if not.

Definition at line 46 of file cdm.c.

```
{   // **scanCopy** Copy ringFrom to ringTo up to but not including string
    uint16_t ch;
    uint8_t j;
    if (str[0] == 0) return 1;   // Null string is always found
    while ((ch = peek(0)) != EOF) {
        if (ch == str[0]) {
            for (j=1; j < 250; j++) {
                if (str[j] == 0) {
                    // Now discard the matching string
                    for (; j>0; j--) get();
                    return 1;  // This is successful match
                }
                if (peek(j) != str[j]) break;
            }
        }
        put(get()); // Copy the ringFrom character to ringTo
    }
    return 0; // String not found
}
```

10.14.4.18 uint16_t scrGet (void)

scrGet Read and remove the next character from scrRing (if any)

If there is an available character in ringSCR remove it from ringSCR and return it. If no character is available return EOF.

Returns

Character if available, otherwise EOF

Definition at line 227 of file cdm.c.

```
{   // **scrGet** Read and remove the next character from scrRing (if any)
    if (scrUsed == 0) return EOF; // Ring is empty
    scrUsed--;
    if (scrGetPtr >= sizeof scrRing) scrGetPtr = 0;
    return scrRing[scrGetPtr++];
}
```

10.14.4.19 uint16_t scrPeek (int16_t *n*)

scrPeek Read character n counting from the beginning or end of scrRing

Parameters

n	Positive or negative index into ringSCR

Returns

Character if available, otherwise EOF.

Definition at line 235 of file cdm.c.

```
{   // **scrPeek** Read character n counting from the beginning or end of scrRing
    uint16_t i;
    if (scrUsed == 0) return EOF; // Ring is empty
    i = iPeekOffset(n, scrGetPtr, scrUsed, sizeof
  scrRing);
    if (i == EOF) return EOF;   // peek past beginning or end of actual data
    return scrRing[i];
}
```

10.14.4.20 uint16_t scrPut (uint16_t *ch*)

scrPut Add the character ch to the end of scrRing

If ch is EOF, simply return EOF. If there is room in ringSCR for another character, add ch to the ring. If the ring is full, set ErrorBits.fscrRingOver = 1 and return EOF.

Parameters

ch	Character to append to ringSCR or EOF

Returns

the same input character ch or EOF

Definition at line 244 of file cdm.c.

```
{   // **scrPut** Add the character ch to the end of scrRing
    if (ch == EOF) return EOF;
    if (scrUsed == sizeof scrRing) {
        ErrorBits.fscrRingOver = 1; // scrRing is full
        return EOF;
    } else {
        if (scrUsed == 0) {scrPutPtr = 0; scrGetPtr = 0; }   // Reset scrRing
        if (scrPutPtr >= sizeof scrRing) scrPutPtr = 0; // wrap putPtr to the beginning
        scrRing[scrPutPtr++] = ch;
        scrUsed++;
    }
    return ch;
}
```

10.14.5 Variable Documentation

10.14.5.1 uint8_t cdRxGetPtr

Index of next character to be read from the PCD-Bus Receive Ring.

Definition at line 386 of file cdm.h.

10.14.5.2 uint8_t cdRxPutPtr

Index of next character to be placed in the PCD-Bus Receive Ring.

Definition at line 387 of file cdm.h.

10.14.5.3 uint8_t cdRxRing[cdRingSize]

Ring Buffer for bytes being received over the PCD-Bus.

Definition at line 366 of file cdm.h.

10.14.5.4 uint8_t cdRxUsed

Bytes used in cdRxRing.

Definition at line 388 of file cdm.h.

10.14.5.5 uint8_t cdTxGetPtr

Index of next character to be sent from the PCD-Bus Transmit Ring.

Definition at line 390 of file cdm.h.

10.14.5.6 uint8_t cdTxPutPtr

Index of next character to be placed in the PCD-Bus Transmit Ring.

Definition at line 391 of file cdm.h.

10.14.5.7 uint8_t cdTxRing[cdRingSize]

Ring Buffer for bytes to be transmitted over the PCD-Bus.

Definition at line 365 of file cdm.h.

10.14.5.8 uint8_t cdTxUsed

Bytes used in cdTxRing.

Definition at line 392 of file cdm.h.

10.14.5.9 uint8_t msgGetPtr

index to get next byte from msgRing

Definition at line 403 of file cdm.h.

10.14.5.10 uint8_t msgPutPtr

Index to put next byte into msgRing.

Definition at line 404 of file cdm.h.

10.14.5.11 uint8_t msgRing[msgRingSize]

Message Ring buffer.

Definition at line 375 of file cdm.h.

10.14.5.12 uint8_t msgUsed

Bytes used in msgRing.

Definition at line 405 of file cdm.h.

10.14.5.13 uint8_t resGetPtr

index to get next byte from resRing

Definition at line 411 of file cdm.h.

10.14.5.14 uint8_t resPutPtr

Index to put next byte into resRing.

Definition at line 412 of file cdm.h.

10.14.5.15 uint8_t resRing[resRingSize]

Response Ring buffer.

Definition at line 381 of file cdm.h.

10.14.5.16 uint8_t resUsed

Bytes used in resRing.

Definition at line 413 of file cdm.h.

10.14.5.17 ringSelect_t ringFrom

Source for get() and peek()

Definition at line 383 of file cdm.h.

10.14.5.18 ringSelect_t ringTo

Destination for put(ch)

Definition at line 384 of file cdm.h.

10.14.5.19 uint8_t rxEcho

Character to echo to the UART Transmit hardware.

Definition at line 397 of file cdm.h.

10.14.5.20 uint8_t rxGetPtr

index to get next byte from rxRing

Definition at line 394 of file cdm.h.

10.14.5.21 uint8_t rxPutPtr

Index to put next byte into rxRing.

Definition at line 395 of file cdm.h.

10.14.5.22 uint8_t rxRing[rxRingSize]

UART Receive Ring buffer.

Definition at line 369 of file cdm.h.

10.14.5.23 uint8_t rxUsed

Bytes used in rxRing.

Definition at line 396 of file cdm.h.

10.14.5.24 uint8_t scrGetPtr

index to get next byte from scrRing

Definition at line 407 of file cdm.h.

10.14.5.25 uint8_t scrPutPtr

Index to put next byte into scrRing.

Definition at line 408 of file cdm.h.

10.14.5.26 uint8_t scrRing[scrRingSize]

Scratch Ring buffer.

Definition at line 378 of file cdm.h.

10.14.5.27 uint8_t scrUsed

Bytes used in scrRing.

Definition at line 409 of file cdm.h.

10.14.5.28 uint8_t txGetPtr

Index of next character to send in the UART Transmit Ring.

Definition at line 399 of file cdm.h.

10.14.5.29 uint8_t txPutPtr

Index of next character to append to the UART Transmit Ring.

Definition at line 400 of file cdm.h.

10.14.5.30 uint8_t txRing[txRingSize]

Ring Buffer data automatically sent to the UART.

Definition at line 372 of file cdm.h.

10.14.5.31 uint8_t txUsed

Bytes used in txRing.

Definition at line 401 of file cdm.h.

10.15 Real-Time Clock

Maintenance and display of the Real-Time Clock and Timers.

Functions

- void rtcTick (void)

  rtcTick *Once-per-second update of the Real Time Clock, Calendar and elapsed timers*
- void rtcSet (uint8_t y, uint8_t m, uint8_t d)

  rtcSet *Set the Local Time calendar in the Real Time Clock*
- void rtcGetSavedParams (void)

  rtcGetSavedParams *Validate RTC EEPROM info and Total Running Time*
- void rtcISR (void)

  rtcISR *8-per-second interrupt for clock and LED blinking*
- uint8_t rtcPoll (void)

  rtcPoll *Clock Polling handles the rather infrequent operations such as calendar updates*
- uint32_t rtcMicroSeconds ()

  rtcMicroSeconds *Returns the current timer value in pseudo-microseconds.*
- void rtcDateTime (uint8_t format)

  rtcDateTime *Make a display of the date and time controlled by select bits in the parameter*

Variables

- uint32_t rtcElapsed

  Running Time in Seconds since power on.

10.15.1 Detailed Description

Maintenance and display of the Real-Time Clock and Timers.

10.15.2 Function Documentation

10.15.2.1 void rtcDateTime (uint8_t *format*)

rtcDateTime Make a display of the date and time controlled by select bits in the parameter

format 0x40 force full formatting format 0x20 force units to be appended format 0x01 integer current days running time (since last reset) format 0x02 d=yymmdd or d=yy-mm-dd format 0x04 t0=nn seconds running time this day format 0x08 t=123456 or t=12:34:56 calendar time today

Parameters

format	Bit Mask controlling the output of the date and time

Definition at line 1275 of file cdm.c.

```
{   // **rtcDateTime** Make a display of the date and time controlled by select bits in the parameter
    if (format == 0) {
        if (ModeBits.fUnits) format |= 0x20;
        if (ModeBits.fFormat) format |= 0x40;
    }
```

```
    if (format & 0x01) {   // Days Running Time
        putInt((rtcElapsed / 86400), 0);    // Elapsed Days
        if (format & 0x20) put('d');
    }
    if (format & 0x02) {   // Year Month Day Calendar
        putInt(rtcYear, 2);
        if (ModeBits.fFormat || (format & 0x40)) put('-');
        putInt(rtcMonth, 2);
        if (ModeBits.fFormat || (format & 0x40)) put('-');
        putInt(rtcDay, 2);
    }
    if (format & 0x04) {   // Seconds Running Time
        if (ModeBits.fFormat || (format & 0x40)) {
            putInt((rtcElapsed / 3600) % 24, 2);
            put(':');
            putInt((rtcElapsed / 60) % 60, 2);
            put(':');
            putInt(rtcElapsed % 60, 2);
        } else {
            putInt((rtcElapsed % 86400), 0);
            if (ModeBits.fUnits || (format & 0x20)) put('s');
        }
    }
    if (format & 0x08) {   // Hours Minutes Seconds Clock Time
        putInt((rtcClock / 3600) % 24, 2);
        if (ModeBits.fFormat || (format & 0x40)) put(':');
        putInt((rtcClock / 60) % 60, 2);
        if (ModeBits.fFormat || (format & 0x40)) put(':');
        putInt(rtcClock % 60, 2);
    }
}
```

10.15.2.2 void rtcGetSavedParams (void)

rtcGetSavedParams Validate RTC EEPROM info and Total Running Time

Definition at line 1189 of file cdm.c.

```
{   // **rtcGetSavedParams** Validate RTC EEPROM info and Total Running Time
    if (eeGet32(nvTRT) > (99 * 365 * 24 * 60 * 60)) {    // Sanity for uninitialized memory
        EEADR = nvTRT; eePut32(0);  // Save null Total Running Time
    }
    if (eeGet32(nvRTO) > (99 * 365 * 24 * 60 * 60)) {    // Sanity for uninitialized memory
        EEADR = nvRTO; eePut32(0);  // Save null clock offset
    }
    if (abs((int16_t)eeGet16(nvZone)) > (12 * 60)) {    // Sanity for uninitialized memory
        EEADR = nvZone; eePut16(0);  // Save null Time Zone
    }
    rtcElapsed = eeGet32(nvTRT);    // Initialize Total Running Time
    FlagBits.fClockSet = 1; // Run Local Time clock setting on next tick
}
```

10.15.2.3 void rtcISR (void)

rtcISR 8-per-second interrupt for clock and LED blinking

Definition at line 1204 of file cdm.c.

```
{   // **rtcISR** 8-per-second interrupt for clock and LED blinking
    uint16_t temp;
    if (INTCONbits.TMR0IF) {
        INTCONbits.TMR0IF = 0; // Clear the interrupt

        temp = TMR0;        // Reads TMR0L then TMR0H which is good for hardware snapshot
        temp -= rtcReload;  // Counter incremented past FFFF so we back it off
        TMR0 = temp;        // Compiler emits: Write TMR0L then TMR0H which is bad
        TMR0 = temp;        // So we do it again to force correct high byte

            // These writes cleared the prescaler, so we actually lose the
            // ISR latency on every tick.  We may need to fudge this back somewhere
```

```
1217            // in case we are actually able to achieve this level of accuracy
1218            // with any of this.
1219
1220        rtcTicks++;     // Increment the sub-second tick count
1221
1222        if (ModeBits.fPollLED == 0) {   // Use the ISR for LED blinking
1223            if (ModeBits.fDiagLED == 0) {
1224                if (ledMask == 0) ledMask = 0x80;
1225                leftLED = (ledA & ledMask) ? 1 : 0;    // Blink the LEDs
1226                rightLED = (ledB & ledMask) ? 1 : 0;
1227                ledMask >>= 1;
1228                if (ledMask == 0) {
1229                    ledMask = 0x80;
1230                    ledA = ledAreload;  // LED blinking is always synchronized
1231                    ledB = ledBreload;  // and uses all 8 bits of the pattern
1232                }
1233            }
1234        }
1235    }
1236 }
```

10.15.2.4 uint32_t rtcMicroSeconds ( )

rtcMicroSeconds Returns the current timer value in pseudo-microseconds.

Read the TMR0 timer and scale the results to microseconds. The RTC interrupt rate is 8 per second, which causes reload of the TMR0. Compensate for this and add the software-maintained running-time elapsed seconds. The result range allows up to about 71 seconds expressed as microseconds.

Returns

Time in microseconds

Definition at line 1238 of file cdm.c.

```
1239 {   // **rtcMicroSeconds** Returns the current timer value in pseudo-microseconds.
1240     uint16_t seconds = rtcElapsed;
1241     uint8_t ticks = rtcTicks;
1242     uint16_t timer = TMR0;
1243     uint32_t micro;
1244     if (ticks != rtcTicks) {    // Timer interrupt just happened and TMR0 was reloaded
1245         ticks = rtcTicks;
1246         timer = TMR0;
1247     }
1248     // timer    range is rtcReload..0xFFFF (5859..65535) over 0.125 seconds
1249     // ticks    range is 0..255 incrementing every 0.125 seconds
1250     // seconds  range is 0..65535 incrementing every second
1251     micro = timer-rtcReload;    // Actual TMR0 cycles (0..65536-rtcReload) per 0.125 sec
1252         // Now scale to microseconds keeping the uint32 from overflowing
1253     micro = (micro * 125000/2) / ((65536-rtcReload)/2);
1254         // Sum Elapsed seconds, interrupt ticks, and scaled TMR0 cycles
1255         // This result wraps around every 71 seconds or so
1256     return seconds*1000000 + ticks*125000 + micro;
1257 }
```

10.15.2.5 uint8_t rtcPoll (void)

rtcPoll Clock Polling handles the rather infrequent operations such as calendar updates

Definition at line 1259 of file cdm.c.

```
1260 {   // **rtcPoll** Clock Polling handles the rather infrequent operations such as calendar updates
1261     if ((rtcTicks & 0xF8) == 0) return 0;
1262     while (rtcTicks & 0xF8) {
1263         rtcTicks -= 0x08;
1264         rtcElapsed++;
```

```
        rtcClock++;
        rtcTick();  // Maintain the Local Date Time structure
        if (rtcTimerA) rtcTimerA--;     // Count down till we reach zero
        if (rtcTimerB) rtcTimerB--;
        if (rtcTimerC) rtcTimerC--;

    }
    return 1;   // Indicates that at least one second tick has occurred
}
```

10.15.2.6 void rtcSet (uint8_t *y*, uint8_t *m*, uint8_t *d*)

rtcSet Set the Local Time calendar in the Real Time Clock

Parameters

y	
m	
d	

Definition at line 1170 of file cdm.c.

```
{   // **rtcSet** Set the Local Time calendar in the Real Time Clock
    uint32_t rtcOffset = 0;
    if (y == 0) y = rtcYear;
    if (m == 0) m = rtcMonth;
    if (d == 0) d = rtcDay;
    if ((y > 99) || (m > 12) || (d > 31)) return;   // Maintain sanity
    EEADR = nvRTO; eePut32(rtcOffset);  // Clear the previous offset
    FlagBits.fClockSet = 1;
    rtcTick();
    while ((rtcYear < y) || (rtcMonth < m) || ((rtcMonth == m) && (
rtcDay < d))) {
        rtcOffset += 24 * 60 * 60;
        rtcClock += 24 * 60 * 60;
        rtcTick();
    }
    EEADR = nvRTO; eePut32(rtcOffset);  // Save the new offset
    FlagBits.fClockSet = 1; // Run Local Time clock setting on next tick
}
```

10.15.2.7 void rtcTick (void)

rtcTick Once-per-second update of the Real Time Clock, Calendar and elapsed timers

Definition at line 1125 of file cdm.c.

```
{   // **rtcTick** Once-per-second update of the Real Time Clock, Calendar and elapsed timers
    uint8_t fNext;
    if (FlagBits.fClockSet) {   // Make all the local stuff be correct
        FlagBits.fClockSet = 0;
        rtcClock = rtcElapsed + eeGet32(nvRTO) + (int16_t)
eeGet16(nvZone) * 60;
        rtcYear = 14;   // Internal Reference Date is 01 Jan 2014
        rtcMonth = 1;
        rtcDay = 1;
        rtcDOW = 4; // Was a Wednesday
    }
    while (rtcClock >= (24 * 60 * 60)) {
        rtcClock -= 24*60*60;
        rtcDay++;
        rtcDOW++; if (rtcDOW > 7) rtcDOW = 1;   // Handle Day of Week

        fNext = 0;     // Not incrementing month
        if (rtcDay > 31) fNext = 1;
        else if (rtcDay > 30) {
            if ((rtcMonth == 4) ||
                (rtcMonth == 6) ||
```

```
                (rtcMonth == 9) ||
                (rtcMonth ==11)) fNext = 1;
        }
        else if (rtcMonth == 2) {
            if ((rtcYear & 3) == 0) {
                if (rtcDay > 29) fNext = 1;
            } else {
                if (rtcDay > 28) fNext = 1;
            }
        }
        if (fNext) {    // Step to next month
            rtcMonth++;
            rtcDay = 1;
            if (rtcMonth > 12) {
                rtcMonth = 1;
                rtcYear++;
            }
        }
    }
    if ((rtcElapsed & 0xFF) == 0) {   // Save the required non-volatile parameters
        EEADR = nvTRT; eePut32(rtcElapsed);
    }
}
```

10.15.3 Variable Documentation

10.15.3.1 uint32_t rtcElapsed

Running Time in Seconds since power on.

Definition at line 1215 of file cdm.h.

10.16 Module Hardware Styles

Handle PCD-Bus Modules with different hardware styles.

Functions

- void initMessage (void)

  initMessage *Power-On Initialization Message identifying the module*
- void Virgin (void)

  Virgin *Module has just been manufactured and has no actual assigned Style*
- void Serial (void)

  Serial *Module is the Manufacturing Serial Interface*
- void Test (void)

  Test *Module is a general-purpose test unit*
- void RS485 (void)

  RS485 *Module has a RS485 Serial Interface*
- void User (void)

  User *Module has a User Interface consisting of an LCD Display and buttons or a Scroll Wheel*
- void USB (void)

  USB *Module implements the on-chip USB Serial Port interface*
- void Laser (void)

  Laser *Module has a Laser Distance Measurement sensor for fluid level measurement*
- uint8_t performLaser (uint8_t ring)

  performLaser *Send commands to the Laser Module and format the results into ring*
- void laserOnTest (void)

  laserOnTest *Send commands to the Laser Module to leave the laser on for 30 seconds*

Variables

- uint8_t opPhase

  Index for the individual Style State Machines.

10.16.1 Detailed Description

Handle PCD-Bus Modules with different hardware styles.

We support the following Hardware Styles:

- User Interface Modules
  - LCD Display and buttons or Scroll Wheel
  - Serial Interface for connection to a terminal for diagnostics and user control
  - USB interface for diagnostics and user control
- Wired Communications Modules
  - RS-485 Multi-Drop Interconnect and MODBUS Interface
- Wireless Communications Modules

  - Bluetooth Short-range Wireless Communication
  - GSM Cellular Radio Modem for two-way SMS messages
  - CDMA Cellular Radio Modem for two-way SMS messages
  - Wi-Fi Local-area Wireless Communication
- Sendor Modules
  - Laser Distance Measurement Sensor for Fluid Level Measurement
  - Ullage Measurement Sensor
- Application-Specific Controller Modules
  - Manufacturing Verification, Test and Serial Number Assignment Module
- Virgin, ininitialized unit from manufacturing

10.16.2 Function Documentation

10.16.2.1 void initMessage (void)

initMessage Power-On Initialization Message identifying the module

// End Styles

Definition at line 2425 of file cdm.c.

```
2426 {   // **initMessage** Power-On Initialization Message identifying the module
2427     sleepLevel(2);  // Turn the power on in case we need the serial port
2428
2429     ringTo = ringTX;
2430     if (ModeBits.fLCD) {
2431         ringTo = ringDisp;
2432         diCursor = 0xFF;     // Clear the display during boot
2433         diPut(EOF);
2434
2435         adcDisplayInterval = 15;
2436     }
2437     putCRLF();
2438     putS("Control/Data (");
2439     copyFromEEs(nvID, 16);
2440
2441     putS(") ");
2442     putInt(eeGet(nvReset), 0);      // Show hardware reset count here
2443     putS(", ");
2444
2445     diCursor = 16;
2446     showVersion(0x0F);
2447     putCRLF();
2448     wait(2);
2449
2450     diCursor = 0xFF;     // Clear display and put clock
2451     put(EOF);
2452 }
```

10.16.2.2 void Laser (void)

Laser Module has a Laser Distance Measurement sensor for fluid level measurement

Definition at line 2574 of file cdm.c.

```
2575 {   // **Laser** Module has a Laser Distance Measurement sensor for fluid level measurement
2576     sleepLevel(1);  // Aux Power Off
2577     FlagBits.fAllowCRLF = 1;
2578
```

```
    ledAreload = 0x00;  // Slow Flash for idle
    ledBreload = 0x01;

    if (resUsed) {
        if (cdState == cdIdle) {    // Reply with any results to the bus
            ringCopy(ringCDtx, ringRES);
            wait(0);
        }
    } else
    if (opPhase) {
        if (opPhase == 0x01) {
            if (performLaser(ringSCR)) ringCopy(ringCDtx, ringSCR);
        }
        if (opPhase == 0x09) laserOnTest();

        opPhase = 0;
    } else
    if (cdState == cdTrmtDone) cdState = cdWaitIdle; else
    if (cdState == cdRecvDone) {
        ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
        doSentence(ringRES, ringCDrx); // Anytime we see something, try to execute it
    } else
    if (rtcTimerA == 0) {
        rtcTimerA = eeGet32(nvInt0);
        EEADR = nvCmd0;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerB == 0) {
        rtcTimerB = eeGet32(nvInt1);
        EEADR = nvCmd1;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerC == 0) {
        rtcTimerC = eeGet32(nvInt2);
        EEADR = nvCmd2;
        doSentence(ringRES, ringEE);
    }
}
```

10.16.2.3 void laserOnTest (void)

laserOnTest Send commands to the Laser Module to leave the laser on for 30 seconds

Definition at line 2395 of file cdm.c.

```
{   // **laserOnTest** Send commands to the Laser Module to leave the laser on for 30 seconds
    sleepLevel(2);
    wait(1);
    ringFrom = ringRX;
    ringTo = ringTX;
    put('S');   // First command; discard results
    wait(1);

    put('O');   // Turn on the laser and leave it on
    wait(30);
    put('C');   // Turn off the laser
    wait(1);
    sleepLevel(1);
}
```

10.16.2.4 uint8_t performLaser (uint8_t *ring*)

performLaser Send commands to the Laser Module and format the results into ring

Parameters

ring	

Returns

Definition at line 2342 of file cdm.c.

```
{   // **performLaser** Send commands to the Laser Module and format the results into ring
    sleepLevel(2);
    wait(1);
    ringFrom = ringRX;
    ringTo = ringTX;
    put('S');   // First command; discard results
    wait(1);
    ringReset(ringFrom);

    put('D');   // Results will look like "D: 0.999m, 0589"
    if (pos(",") == EOF) wait(1);
    if (pos(",") == EOF) wait(1);
    if (pos(",") == EOF) wait(1);
    if (pos(",") == EOF) wait(1);
    put('S');   // Results will look like "S: 29.2`C,3.0V"
    wait(1);
    sleepLevel(1);  // Turn off the laser when we are done

    // Now parse the results into a response message
    ringTo = ring;
    ringReset(ringTo);

    putS("S:"); putSix(eeGet32(nvSN));  // Make a message from me

    if (pos("m") != EOF) {  // Expect "D: 1.234m"
        putS(";D0:");
        scan(" ");
        scanCopy("m");
    }
    if (pos("E") != EOF) {  // Expect "D:Er09!"
        putS(";E0:");
        scan(":");
        scanCopy("!");
    }

    scan("S:");             // Expect "S: 29.2`C,3.0V"
    if (pos("C") != EOF) {
        putS(";T:");
        scan(" ");
        scanCopy("\x60" "C");     // Laser says `C for units on temperature
    }
    if (pos("V") != EOF) {
        putS(";a9:");
        scan(",");
        scanCopy("V");
    }

    putS(";a0:");
    adcShowChannel(0);      // Just for the hell of it, include our supply voltage
    return 1;
}
```

10.16.2.5 void RS485 (void)

RS485 Module has a RS485 Serial Interface

Definition at line 2748 of file cdm.c.

```
{   // **RS485** Module has a RS485 Serial Interface
    sleepLevel(2);  // Turn the power on in case we need the serial port
    FlagBits.fAllowCRLF = 1;
```

```
    ledAreload = 0x00;  // Slow Single Flash for Serial Board
    ledBreload = 0x01;

    ringFrom = ringRX;
    if (peek(-1) == 0x0D) { // Wait for a CR
        if (cdState == cdIdle) {
            ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for 485 Received
            ringCopy(ringCDtx, ringRX);
            wait(0);
        }
    }

    if (cdState == cdTrmtDone) cdState = cdWaitIdle;
    if (cdState == cdRecvDone) {    // Honor any commands we get from the CD-Bus
        ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
        if (checkSum(ringCDrx) <= 0xFF) {
            doSentence(ringRES, ringCDrx);
        } else ringReset (ringCDrx);
    }

    if (resUsed) {
        if (cdState == cdIdle) {    // Reply with any results to the bus
            ringCopy(ringCDtx, ringRES);
            wait(0);
        }
    }
}
```

10.16.2.6 void Serial (void)

Serial Module is the Manufacturing Serial Interface

This is essentially a CD-Bus to Serial adapter.

Used primarily for development and manufacturing.

Allows configuration commands to be sent to freshly-programmed, Virgin devices.

Definition at line 2490 of file cdm.c.

```
{   // **Serial** Module is the Manufacturing Serial Interface
    sleepLevel(2);  // Turn the power on in case we need the serial port
    FlagBits.fAllowCRLF = 1;

    ledAreload = 0x00;  // Slow Single Flash for Serial Board
    ledBreload = 0x01;

    ringFrom = ringRX;
    if (peek(-1) == 0x0D) { // Wait for a CR
        ledMask = 0; ledA = 0x00;  ledB = 0xF0;  // ledB for CD-Bus Transmit
        if (checkSum(ringRX) <= 0xFF) {
            doSentence(ringRES, ringRX);
            if (resUsed > 0) {
                ringTo = ringTX;
                putS("Result: ");
                ringCopy(ringTX, ringRES);
                putCRLF();
            }
        }
        ringReset (ringRX);
    }

    if (cdState == cdTrmtDone) cdState = cdWaitIdle;
    if (cdState == cdRecvDone) {    // Display anything we get from the CD-Bus
        ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
        ringTo = ringTX;
            // DO NOT honor commands from the CD-Bus
        putS("s--");
        ringCopy(ringTX, ringCDrx);     // The cdState goes cdWaitIdle when this
finishes
        putCRLF();
    }
}
```

10.16.2.7 void Test (void)

Test Module is a general-purpose test unit

Provides a test platform for different features during development

Definition at line 2682 of file cdm.c.

```
{   // **Test** Module implements Bus Diagnostics
    sleepLevel(2);  // Aux Power Off
    FlagBits.fAllowCRLF = 1;

    ledAreload = 0x00;  // Slow Flash for idle
    ledBreload = 0x01;

    if (rxUsed) { // Wait for something from the serial port and maybe a receive timeout
        if ((SerialBits.fTimeouts == 0) ||
            ((rtcMicroSeconds()-rxMicroSeconds) > 2000)) {
            ledMask = 0; ledA = 0x00;  ledB = 0xF0;  // ledB for CD-Bus Transmit
            if (cdState == cdIdle) {
                ringTo = ringCDtx;
                putS("@@");
                FlagBits.fNeedID = 1;
                putFieldDelimiter();
                ringCopy (ringCDtx, ringRX);    // Test Mode: Let the world know what I got
            }
        }
    }

    if (resUsed) {
        if (cdState == cdTrmtDone) cdState = cdWaitIdle;
        if (cdState == cdRecvDone) cdState = cdWaitIdle;
        if (cdState == cdIdle) {    // Reply with any results to the bus
            ringCopy(ringCDtx, ringRES);
        }
    } else
    if (opPhase) {
        if (opPhase == 0x02) {
            ledMask = 0; ledA = 0xA0;
            ringTo = ringCDtx;
            putS("Hello");
            wait(1);
            putS("There");
        }
        opPhase = 0;
    } else
    if (cdState == cdTrmtDone) cdState = cdWaitIdle; else
    if (cdState == cdRecvDone) {
        ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
        doSentence(ringRES, ringCDrx); // Anytime we see something, try to execute it
    } else
    if (rtcTimerA == 0) {
        rtcTimerA = eeGet32(nvInt0);
        EEADR = nvCmd0;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerB == 0) {
        rtcTimerB = eeGet32(nvInt1);
        EEADR = nvCmd1;
        doSentence(ringRES, ringEE);
    } else
    if (rtcTimerC == 0) {
        rtcTimerC = eeGet32(nvInt2);
        EEADR = nvCmd2;
        doSentence(ringRES, ringEE);
    }
}
```

10.16.2.8 void USB (void)

USB Module implements the on-chip USB Serial Port interface

Definition at line 2664 of file cdm.c.

```
{   // **USB** Module implements the on-chip USB Serial Port interface

    sleepLevel(2);  // Aux Power Off
    FlagBits.fAllowCRLF = 1;

    ledAreload = 0x00;  // Slow Flash for idle
    ledBreload = 0x01;

    if ((rtcTimerA & 0x0F) == 0) { // Every 16 seconds send a serial port message
        ledMask = 0; ledB = 0xF0;
        ringTo = ringTX;
        putS("Testing");
        putSix(eeGet32(nvSN));  // My Serial Number
        wait(1);
    }
}
```

10.16.2.9 void User (void)

User Module has a User Interface consisting of an LCD Display and buttons or a Scroll Wheel

Definition at line 2618 of file cdm.c.

```
{   // **User** Module has a User Interface consisting of an LCD Display and buttons or a Scroll Wheel
    sleepLevel(2);
    ringTo = ringTX;
    adcDisplayInterval = 15;

    ledAreload = 0x00;  // Slow Flash for idle
    ledBreload = 0x01;

    while(1) {
        if (wait(0)) {
            if (adcDisplayInterval) if ((rtcElapsed %
adcDisplayInterval) == 0) {
                ringTo = ringDisp; diCursor = 24;
                adcShow(0);    // Show only supply voltage
                diBlankField(31);
            }
        }

        if (uiChanged || uiHeld) {
            ringTo = ringDisp; diCursor = 24;

            if (uiCurrent & uiUp) put('u'); else put('U');
            if (uiCurrent & uiEnter) put('e'); else put('E');
            if (uiCurrent & uiDown) put('d'); else put('D');
            put('-');
            putInt(uiScroll, 0);
            diBlankField(31);
            ModeBits.fUIclock = 1;  // Turn the clock display back on in case it was off
        }

        if (cdState == cdTrmtDone) cdState = cdWaitIdle;
        if (cdState == cdRecvDone) {
            ringFrom = ringCDrx;
            ringTo = ringDisp; diCursor = 0xFF;
            ledMask = 0; ledA = 0xF0;  ledB = 0x00;  // ledA for CD-Bus Received
            while (put(get()) != EOF) continue;
            // Show everything received on the CDbus on the display
            // Getting the last character releases cdState back to cdWaitIdle
            diBlankField(31);
            ModeBits.fUIclock = 0;  // Turn off the clock display for a while
        }

        menu();
    }
}
```

10.16.2.10 void Virgin (void)

Virgin Module has just been manufactured and has no actual assigned Style

Definition at line 2454 of file cdm.c.

```
{   // **Virgin** Module has just been manufactured and has no actual assigned Style
    sleepLevel(2);  // Turn the power on in case we need the serial port
    FlagBits.fAllowCRLF = 1;

    ledAreload = 0xAA;  // Double fast-flash for Virgin Mode
    ledBreload = 0x55;

    ringFrom = ringRX;
    if (peek(-1) == 0x0D) { // Wait for a CR
        ledMask = 0; ledA = 0x00;  ledB = 0xF0;  // ledB for CD-Bus Transmit
        if (checkSum(ringRX) <= 0xFF) {
            doSentence(ringRES, ringRX);
            if (resUsed > 0) {
                ringTo = ringTX;
                putS("Command: ");
                ringCopy(ringTX, ringRES);
                putCRLF();
            }
        }
        ringReset (ringRX);
    }

    if (cdState == cdTrmtDone) cdState = cdWaitIdle;
    if (cdState == cdRecvDone) {
        ledMask = 0; ledA = 0xFF;  ledB = 0xFF;  // ledA for CD-Bus Received

        doSentence(ringRES, ringCDrx);   // Anytime we see something, try to execute it
        if (resUsed > 0) {
            wait(1);
            ringCopy(ringCDtx, ringRES);
            wait(1);    // This sends the results out the CD-Bus
        }
    }
}
```

10.16.3 Variable Documentation

10.16.3.1 uint8_t opPhase

Index for the individual Style State Machines.

Definition at line 1526 of file cdm.h.

10.17 UART Communication

Handle Serial I/O to a variety of peripheral devices.

Data Structures

- struct SerialBits_t

  SerialBits_t *USART Control Bits*

Enumerations

- enum allowedBaud_t {
  baud1200, baud2400, baud4800, baud9600,
  baud19200, baud57600, baud115200, baudSpecial }

  allowedBaud_t *Allowed Baud Rates for the UART*

Functions

- uint16_t rxGet (void)

  rxGet *Read and remove the next character from rxRing (if any).*
- uint16_t rxPeek (int16_t n)

  rxPeek *Read character n counting from the beginning or end of rxRing*
- void rxPoll (void)

  rxPoll *Poll the UART Hardware receiver and get chars into rxRing*
- uint16_t txPut (uint16_t ch)

  txPut *Add a character to the Serial Port Transmit Ring Buffer txRing*
- void txPoll (void)

  txPoll *Poll the UART Hardware Transmitter and send from txRing*
- void ensureBaud (allowedBaud_t baud)

  ensureBaud *Safely ensure that the correct baud rate divisor is set in hardware*

10.17.1 Detailed Description

Handle Serial I/O to a variety of peripheral devices.

10.17.2 Enumeration Type Documentation

10.17.2.1 enum allowedBaud_t

allowedBaud_t Allowed Baud Rates for the UART

Enumerator

baud1200 0

baud2400 1

baud4800 2

baud9600 3

baud19200 4

baud57600 5

baud115200 6

baudSpecial 7

Definition at line 190 of file cdm.h.

```
191 {   // **allowedBaud_t** Allowed Baud Rates for the UART
192     baud1200,
193     baud2400,
194     baud4800,
195     baud9600,
196     baud19200,
197     baud57600,
198     baud115200,
199     baudSpecial
200 } allowedBaud_t;
```

10.17.3 Function Documentation

10.17.3.1 void ensureBaud (allowedBaud_t *baud*)

ensureBaud Safely ensure that the correct baud rate divisor is set in hardware

This can be called anytime to ensure that the USART is set at the desired baud rate.

It only touches the hardware if a change needs to be made.

Parameters

baud	Index of allowedBaud_t, NOT a numeric baud rate!

Definition at line 186 of file cdm.c.

```
187 {   // **ensureBaud** Safely change the Baud Rate Generator divisor only if necessary.
188     uint16_t divisor = 0;   // Numeric Divisor for Baud Rate Generator
189     uint8_t fSPEN;          // Current state of Serial Port Enble
190         // These divisors assume we are running at 48.0 MHz
191     switch (baud) {
192         case baud115200:    divisor =   25; break;  // This is 115,200 baud
193         case baud57600:     divisor =   51; break;  // This is  57,600 baud
194         case baud19200:     divisor =  155; break;  // This is  19,200 baud
195         case baud9600:      divisor =  312; break;  // This is    9600 baud
196         case baud4800:      divisor =  624; break;  // This is    4800 baud
197         case baud2400:      divisor = 1249; break;  // This is    2400 baud
198         case baud1200:      divisor = 2499; break;  // This is    1200 baud
199         case baudSpecial:   ;
200             // This is for any special cases that may be required
201     }
202
203     if (divisor == 0) return;
204     if (SPBRG1 == (divisor & 0xFF)) return; // Already set. Don't change anything
205
206     // The baud rate changed. We reset the UART and clear any crap.
207     fSPEN = RCSTA1bits.SPEN ? 1 : 0;    // Save the current state of the SPEN
208
209     RCSTA1bits.SPEN = 0; // Disable Serial Port and pin drivers
210     TXSTA1bits.TXEN = 0; // Disable transmitter
211     RCSTA1bits.CREN = 0; // Disable continuous receive, which clears overrun errors
212     if (RCREG1 == RCREG1);  // Read characters to flush the receiver
213
214     SPBRGH1 = divisor / 256;
215     SPBRG1 = divisor & 0xFF;
216
217     BAUDCON1bits.BRG16 = 1; // Sixteen-bit baud rate generator
218     TXSTA1bits.BRGH = 0; // High-speed BRG counter not used
219     BAUDCON1bits.CKTXP = 0; // Normal Polarity Output
220     TXSTA1bits.SYNC = 0; // Async characters
221
```

```
    if (fSPEN) RCSTA1bits.SPEN = 1; // Restore Serial Port and pin drivers
        // Reception(CREN) will be enabled when we poll it
        // Transmission (TXEN) will be enabled when we poll it
}
```

10.17.3.2 uint16_t rxGet (void)

rxGet Read and remove the next character from rxRing (if any).

Returns

Definition at line 413 of file cdm.c.

```
{   //  **rxGet** Read and remove the next character from rxRing (if any).
    rxPoll();   // Explicitly poll here so a tight loop will work
    if (rxUsed == 0) return EOF;    // Ring is empty
    rxUsed--;
    if (rxGetPtr >= sizeof rxRing) rxGetPtr = 0;    // getPtr wraps around
    return rxRing[rxGetPtr++];
}
```

10.17.3.3 uint16_t rxPeek (int16_t *n*)

rxPeek Read character n counting from the beginning or end of rxRing

Parameters

n	

Returns

Definition at line 422 of file cdm.c.

```
{   // **rxPeek** Read character n counting from the beginning or end of rxRing
    uint16_t i;
    if (rxUsed == 0) return EOF;    // Ring is empty
    i = iPeekOffset(n, rxGetPtr, rxUsed, sizeof rxRing);
    if (i == EOF) return EOF;   // peek past beginning or end of actual data
    return rxRing[i];
}
```

10.17.3.4 void rxPoll (void)

rxPoll Poll the UART Hardware receiver and get chars into rxRing

Handle all Serial reception from the EUSART.

We correctly honor all combinations of the control bits in SerialBits. SerialBits.fParity // 1=Enable, 0=Disable Parity Bit SerialBits.fEven // 1=Expect Even parity if fParity is enabled SerialBits.fSevenBit // 0=Eight Bit, 1=Seven Bit SerialBits.↩ fEchoRx // 1=Echo all (non-null) characters back to the TX

The received character is placed in rxRing unless the ring is full.

Status is reported in the ErrorBits. ErrorBits.fRxFraming // 1=There has been a framing error ErrorBits.fRxOver // 1=There has been a hardware overrun ErrorBits.fRxRingOver // 1=There has been a Ring Buffer Full ErrorBits.fRxParity // 1=There has been a Receive Parity Error

On every poll we ensure that the Continuous Receive Enable bit is set.

On every reception we ensure that the hardware 8/9 bit character mode is correct. RCSTA1bits.RX9 = SerialBits.fParity && !SerialBits.fSevenBit;

Definition at line 361 of file cdm.c.

```
{   // **rxPoll** Poll the UART Hardware receiver and get chars into rxRing
    uint8_t ch;
    if (RCSTA1bits.OERR) {
        RCSTA1bits.CREN = 0;    // Clear the hardware error by clear/set CREN
        ErrorBits.fRxOver = 1;  // Indicate the RX Overrun error has occurred
    }
    RCSTA1bits.CREN = 1;
    while (PIR1bits.RC1IF) {
        if (RCSTA1bits.FERR) {
            ch = RCREG1;   // Clear hardware framing errors and ignore character
            ErrorBits.fRxFraming = 1;   // Indicate RX Framing error if anybody cares
        } else {
            if (rxPutPtr >= sizeof rxRing) rxPutPtr = 0;
            if (SerialBits.fParity) {
                if (SerialBits.fSevenBit) {
                    ch = RCREG1;
                    if (parity(ch) != ((SerialBits.fEven ? 0 : 1)))
    ErrorBits.fRxParity = 1;
                } else {
                    if (RCSTA1bits.RX9D) {  // Test the ninth bit before reading the data byte
                        ch = RCREG1;
                        if (parity(ch) == ((SerialBits.fEven ? 0 : 1)))
    ErrorBits.fRxParity = 1;
                    } else {
                        ch = RCREG1;
                        if (parity(ch) != ((SerialBits.fEven ? 0 : 1)))
    ErrorBits.fRxParity = 1;
                    }
                }
            } else ch = RCREG1;    // Just read and save the hardware buffer

            if (SerialBits.fSevenBit) ch &= 0x7F;   // Mask to 7-bit char if needed

                // Ensure that we receive 9-bit characters when we need them
            RCSTA1bits.RX9 = SerialBits.fParity && !SerialBits.
    fSevenBit;

            if (rxUsed < sizeof rxRing) {
                rxRing[rxPutPtr++] = ch; // Add the character to the ring buffer
                rxUsed++;
                if (SerialBits.fEchoRx) rxEcho = ch;  // Echo it, if desired
            } else ErrorBits.fRxRingOver = 1;   // Indicate Receive Ring Overrun

            if (SerialBits.fTimeouts) {
                rxMicroSeconds = rtcMicroSeconds(); // Time character received
     from EUSART Receive Buffer
            }

            if (SerialBits.fFlow) {     // If Hardware Flow Control is enabled
        //        RFRouttris = 0;
        //        RFRout = rxMax < rxUsed;    // Ready For Receive if there is room in the Ring
            }
        }
    }
}
```

10.17.3.5 void txPoll (void)

txPoll Poll the UART Hardware Transmitter and send from txRing

This will NEVER overflow the txRing. It waits for space if needed.

Definition at line 453 of file cdm.c.

```
{   // **txPoll** Poll the UART Hardware Transmitter and send from txRing
    uint8_t ch;
    TXSTA1bits.TXEN = 1; // Always ensure Enable transmitter

        // Ensure that we send 9-bit characters when we need them
    TXSTA1bits.TX9 = SerialBits.fParity && !SerialBits.fSevenBit;

    while (txUsed || rxEcho) {  // Send as many chars as we can buffer up
        if (!PIR1bits.TXIF) break;     // Done if hardware is busy

        if (SerialBits.fFlow) {     // If Hardware Flow Control is enabled
            CTSintris = 1;
            if (CTSin == 0) break;          // Done if he is not Clear To Send
        }

        if (rxEcho) {   // Echo characters are priority over chars from txRing
                        // We ensure echo is with correct parity
            ch = rxEcho;
            rxEcho = 0;
        } else {
            if (SerialBits.fInhibitTX) break;   // Do not transmit from ring if we are inhibited
            if (txGetPtr >= sizeof txRing) txGetPtr = 0;
            ch = txRing[txGetPtr++];    // Next character from txRing
            txUsed--;
        }

        if (SerialBits.fSevenBit) ch &= 0x7F;   // Mask to 7-bit if needed

        if (SerialBits.fParity) {   // compute parity if needed
            if (SerialBits.fSevenBit) {
                if (parity(ch)) ch |= 0x80;
            } else {
                TXSTA1bits.TX9D = parity(ch);
            }
        }

        TXREG1 = ch;    // Finally start the character transmission

        if (SerialBits.fTimeouts) {
//                txMicroSeconds = rtcMicroSeconds(); // Time character sent to EUSART Transmit Buffer
        }
    }

    if (SerialBits.f485) {
        RTSout = !TXSTAbits.TRMT; // Turn on the RS-485 driver any time we are sending
        RTSouttris = 0;
    }
}
```

10.17.3.6 uint16_t txPut (uint16_t *ch*)

txPut Add a character to the Serial Port Transmit Ring Buffer txRing

Parameters

ch	

Returns

Definition at line 431 of file cdm.c.

```
{   // **txPut** Add a character to the Serial Port Transmit Ring Buffer txRing
    if (ch != EOF) {
        // Prevent ring overrun by waiting until there is space
        // Comment this out if we want the fTxRingOver flag instead
        //while (txUsed >= sizeof txRing) pollCritical();

        // Ignore character if it would overrun the ring.
```

```
        if (txUsed >= sizeof txRing) {
            ErrorBits.fTxRingOver = 1;
            return EOF;
        }

        if (txUsed == 0) {txPutPtr = 0; txGetPtr = 0; }   // Reset txRing
        if (txPutPtr >= sizeof txRing) txPutPtr = 0;
        txRing[txPutPtr++] = ch;
        txUsed++;
    }
    txPoll();   // Explicitly poll here so a tight loop will work
    return ch;
}
```

10.18 User Interface

Provides user interaction via a LCD Display and buttons or scroll wheel.

Functions

- uint16_t diPut (uint16_t ch)

  diPut *Put a character in the LCD Display at locction diCursor*
- void diPoll (void)

  diPoll *Send contents of diDisplay buffer to LCD Display as needed*
- void diBlankField (uint8_t n)

  diBlankField *Blank to the end of the field ending at position n*
- void uiPoll (void)

  uiPoll *Poll the buttons on the user interface to detect bounce and held-down*

Variables

- uint8_t diTicks

  Fast Tick last time the display was updated.

- enum uiState_t {
  none, up, down, enter,
  back, holding }

  uiState_t *Allowed States for Buttons or Scroll Wheel*

10.18.1 Detailed Description

Provides user interaction via a LCD Display and buttons or scroll wheel.

10.18.2 Enumeration Type Documentation

10.18.2.1 enum uiState_t

uiState_t Allowed States for Buttons or Scroll Wheel

Enumerator

none All buttons are up.

up Up button is pressed.

down Down button is pressed.

enter Enter button is pressed.

back (virtual) Back button was pressed

holding Button is being held down (i.e. auto-repeating)

Definition at line 280 of file cdm.h.

```
{   // **uiState_t** Allowed States for Buttons or Scroll Wheel
    none,
    up,
    down,
    enter,
    back,
    holding
} uiState_t;
```

10.18.3 Function Documentation

10.18.3.1 void diBlankField (uint8_t *n*)

diBlankField Blank to the end of the field ending at position n

Parameters

n	

Definition at line 1022 of file cdm.c.

```
{   // **diBlankFied** Blank to the end of the field ending at position n
    uint8_t cursor = diCursor;      // save the current cursor position
    n = n & (sizeof diDisplay - 1);
    while (diCursor <= n) diPut(0x20); // Blank remaining characters in the field
    diCursor = cursor;              // Restore the cursor
}
```

10.18.3.2 void diPoll (void)

diPoll Send contents of diDisplay buffer to LCD Display as needed

Definition at line 983 of file cdm.c.

```
{   // **diPoll** Send contents of diDisplay buffer to LCD Display as needed
    uint8_t i;
    if (!ModeBits.fUI) return;
    if (!FlagBits.fChanged) {
        if (diTicks != rtcTicks) {
            diTicks = rtcTicks;
            if (ModeBits.fBlink) FlagBits.fChanged = 1; // only blink on a fast tick
        }
    }
    if (!FlagBits.fChanged) return;

        // Make sure the previous display is complete before doing another
    if (txUsed > 0) return;

    FlagBits.fChanged = 0;
    if (!ModeBits.fLCD) {  // Output to a serial terminal (not LCD)
        ModeBits.fBlink = 0;    // Blinking to a serial port is not sane
        putCRLF();
        for (i=0; i < sizeof diDisplay; i++) txPut(diDisplay[i]);
        return;
    }
        // Refresh the entire LCD Display
    txPut(0xFE); txPut(0x45); txPut(0x00); //Cursor Start Line 1 (home)
    for (i=0; i < (sizeof diDisplay / 2); i++) txPut(diDisplay[i]);
    txPut(0xFE); txPut(0x45); txPut(0x40); //Cursor Start Line 2
    for (; i < sizeof diDisplay; i++) txPut(diDisplay[i]);

    FlagBits.fBlinkOn = (rtcTicks & 4) == 0;
    if (ModeBits.fBlink && !FlagBits.fBlinkOn) return;
        // Position the user entry cursor if needed
    i = uiCursor;
    if (i >= sizeof diDisplay) return;
    if (i >= (sizeof diDisplay / 2)) i += 0x40 - (sizeof diDisplay / 2);
    txPut(0xFE); txPut(0x45); txPut(i); // Cursor Position
```

```
    if (ModeBits.fBlock) txPut(0xFF); // Block Character at cursor position
    if (ModeBits.fUnder) txPut(0x5F); // Underscore Character at cursor position
}
```

10.18.3.3 uint16_t diPut (uint16_t *ch*)

diPut Put a character in the LCD Display at locction diCursor

Parameters

ch	

Returns

Definition at line 961 of file cdm.c.

```
{   // **diPut** Put a character in the LCD Display at locction diCursor
    if (diCursor == 0xFF) {
        for (diCursor=0; diCursor < sizeof diDisplay; diCursor++) {
            if (diDisplay[diCursor] != 0x20) {
                diDisplay[diCursor] = 0x20;   // Blank the display
                FlagBits.fChanged = 1;
            }
        }
        diCursor = 0;
    }
    if (ch != EOF) {
        diCursor = diCursor & (sizeof diDisplay - 1);   // Wrap around
        if (ch != diDisplay[diCursor]) {
            diDisplay[diCursor] = ch;
            FlagBits.fChanged = 1;
        }
        diCursor++;
    }
    return ch;
}
```

10.18.3.4 void uiPoll (void)

uiPoll Poll the buttons on the user interface to detect bounce and held-down

Debounce the buttons as needed Detect the difference between buttons and the scroll wheel.

Definition at line 1030 of file cdm.c.

```
{   // **uiPoll** Poll the buttons on the user interface to detect bounce and held-down
    if (!ModeBits.fUI) return;
    if (uiState != holding) uiState = none;  // clear any previous operation
    if (uiTicks == rtcTicks) return;  // Only run this 8 times per second
    uiTicks = rtcTicks;

    TRISC |= uiButtonMask;  // make sure that the button bits are inputs

    uiState = none;     // clear any previous unprocessed operation

    uiCurrent = uiButtons & uiButtonMask;

    if (uiCurrent == 0) {   // All buttons down. Invalid hardware state
        uiChanged = 0;
        uiState = none;     // No buttons to report
        return;
    }

```

```
    uiChanged = uiCurrent ^ uiLast;
    uiLast = uiCurrent;
    if (uiEnter & uiChanged) {
        if (uiEnter & uiCurrent) {    // We just released the Enter button
            if (uiHeld > 0x1F) uiState = back; else uiState = enter;
        }
    }
    if ((uiEnter & uiCurrent) == 0) uiState = holding;
    if ((uiCurrent & (uiUp | uiDown)) == 0) FlagBits.fWheel = 1;  // Detect Scroll Wheel
    if (FlagBits.fWheel) {   // Quadrature for Up and Down
                // Count duration of Enter button being held
        if (uiCurrent & uiEnter) uiHeld = 0; else uiHeld++;

            // !!! Handle quadrature changes here.  !!! Not Implemented

    } else {    // Simple Buttons for Up and Down, with hold for rapid
                // Count duration of any button being held
        if (uiCurrent == uiButtonMask) uiHeld = 0; else uiHeld++;
        if ((uiUp & uiCurrent) == 0) {
            if ((uiHeld & 0x07) == 1) {   // Repeat 2 per sec
                uiScroll++;
                uiState = up;
            }
        }
        if ((uiDown & uiCurrent) == 0) {
            if ((uiHeld & 0x07) == 1) {   // Repeat 2 per sec
                uiScroll--;
                uiState = down;
            }
        }
    }
}
```

10.18.4 Variable Documentation

10.18.4.1 uint8_t diTicks

Fast Tick last time the display was updated.

Definition at line 1133 of file cdm.h.

10.19 The Interrupt Service Routines

Functions

- void interrupt high_priority isr ()

  high_priority *High Priority Interrupt*

10.19.1 Detailed Description

We have only a single level of interrupt service. Most of the module operations are handled in a polling loop.

See also

pollCritical()

10.19.2 Function Documentation

10.19.2.1 void interrupt high_priority isr ( )

high_priority High Priority Interrupt

Handle the USB interface and the Real-Time Clock. Everything else is done by polling.

Definition at line 161 of file main.c.

```
{   // **high_priority** ...
    rtcISR();
}
```

10.20 USB Interface Public API

Modules

- Descriptor Items

  Items defined by the application which are involved in the enumeration of the device.
- Static Callbacks

  Optional static callback macros to be defined in the application's usb_config.h.
- USB CDC Class Enumerations and Descriptors

  Packet structs, constants, and callback functions implementing the "Universal Serial Bus Class Definitions for Communication Devices" (commonly the USB CDC Specification), version 1.1.

Macros

- #define usb_is_configured() (usb_get_configuration() != 0)

  Determine whether the device is in the Configured state.

Typedefs

- typedef void(∗ usb_ep0_data_stage_callback)(bool transfer_ok, void ∗context)

  Endpoint 0 data stage callback definition.

Functions

- void usb_init (void)

  Initialize the USB library and hardware.
- void usb_service (void)

  Update the USB library and hardware.
- uint8_t usb_get_configuration (void)

  Get the device configuration.
- unsigned char ∗ usb_get_in_buffer (uint8_t endpoint)

  Get a pointer to an endpoint's input buffer.
- void usb_send_in_buffer (uint8_t endpoint, size_t len)

  Send an endpoint's IN buffer to the host.
- bool usb_in_endpoint_busy (uint8_t endpoint)

  Check whether an IN endpoint is busy.
- bool usb_in_endpoint_halted (uint8_t endpoint)

  Check whether an endpoint is halted.
- bool usb_out_endpoint_has_data (uint8_t endpoint)

  Check whether an OUT endpoint has received data.
- void usb_arm_out_endpoint (uint8_t endpoint)

  Re-enable reception on an OUT endpoint.
- bool usb_out_endpoint_halted (uint8_t endpoint)

  Check whether an OUT endpoint is halted.
- uint8_t usb_get_out_buffer (uint8_t endpoint, const unsigned char ∗∗buffer)

  Get a pointer to an endpoint's OUT buffer.

- void usb_start_receive_ep0_data_stage (char ∗buffer, size_t len, usb_ep0_data_stage_callback callback, void ∗context)

  Start the data stage of an OUT control transfer.
- void usb_send_data_stage (char ∗buffer, size_t len, usb_ep0_data_stage_callback callback, void ∗context)

  Start the data stage of an IN control transfer.

10.20.1 Detailed Description

10.20.2 Macro Definition Documentation

10.20.2.1 #define usb_is_configured( ) (usb_get_configuration() != 0)

Determine whether the device is in the Configured state.

Return whether the device is in the configured state. During enumeration, the device will start at the DEFAULT state, transition through ADDRESS, and eventually reach CONFIGURED. The host can also command the device out of the configured state (and back into ADDRESS). The application shouldn't use any of the endpoints unless in the CONFI↩GURED state.

See also

usb_get_configuration()

Definition at line 387 of file usb.h.

10.20.3 Typedef Documentation

10.20.3.1 typedef void(∗ usb_ep0_data_stage_callback)(bool transfer_ok, void ∗context)

Endpoint 0 data stage callback definition.

This is the callback function type expected to be passed to `usb_start_receive_ep0_data_stage()` and `usb_send_data_stage()`. Callback functions will be called by the stack when the event for which they are registered occurs.

Parameters

transfer_ok	*true* if transaction completed successfully, or *false* if there was an error
context	A pointer to application-provided context data

Definition at line 508 of file usb.h.

10.20.4 Function Documentation

10.20.4.1 void usb_arm_out_endpoint (uint8_t *endpoint*)

Re-enable reception on an OUT endpoint.

Re-enable reception on the specified endpoint. Call this function after `usb_out_endpoint_has_data()` indicated that there was data available, and after the application has dealt with the data. Calling this function gives the specified OUT endpoint's buffer back to the USB stack to receive the next transaction.

Parameters

endpoint	The endpoint requested

Definition at line 1507 of file usb.c.

```
{
#ifdef PPB_EPn
    uint8_t ppbi = (ep_buf[endpoint].flags & EP_RX_PPBI)? 1: 0;
    uint8_t pid = (ep_buf[endpoint].flags & EP_RX_DTS)? 1: 0;

    if (pid)
        SET_BDN(BDSnOUT(endpoint,ppbi),
            BDNSTAT_UOWN|BDNSTAT_DTSEN|BDNSTAT_DTS,
            ep_buf[endpoint].out_len);
    else
        SET_BDN(BDSnOUT(endpoint,ppbi),
            BDNSTAT_UOWN|BDNSTAT_DTSEN,
            ep_buf[endpoint].out_len);

    /* Alternate the PPBI */
    ep_buf[endpoint].flags ^= EP_RX_PPBI;
    ep_buf[endpoint].flags ^= EP_RX_DTS;

#else
    uint8_t pid = (ep_buf[endpoint].flags & EP_RX_DTS)? 1: 0;
    if (pid)
        SET_BDN(BDSnOUT(endpoint,0),
            BDNSTAT_UOWN|BDNSTAT_DTS|BDNSTAT_DTSEN,
            ep_buf[endpoint].out_len);
    else
        SET_BDN(BDSnOUT(endpoint,0),
            BDNSTAT_UOWN|BDNSTAT_DTSEN,
            ep_buf[endpoint].out_len);

    ep_buf[endpoint].flags ^= EP_RX_DTS;
#endif

}
```

10.20.4.2 uint8_t usb_get_configuration (void)

Get the device configuration.

Get the device configuration as set by the host. If the device is not in the CONFIGURED state, 0 will be returned.

See also

usb_is_configured()

Returns

Return the device configuration or 0 if the device is not configured.

Definition at line 1406 of file usb.c.

```
{
    return g_configuration;
}
```

10.20.4.3 unsigned char* usb_get_in_buffer (uint8_t *endpoint*)

Get a pointer to an endpoint's input buffer.

This function returns a pointer to an endpoint's input buffer. Call this to get a location to copy IN data to in order to send it to the host. Remember that IN data is data which goes from the device to the host. The maximum length of this buffer is defined by the application in usb_config.h (eg: `EP_1_IN_LEN`). It is wise to call `usb_in_endpoint_busy()` before calling this function.

Parameters

endpoint	The endpoint requested

Returns

Return a pointer to the endpoint's buffer.

Definition at line 1411 of file usb.c.

```
{
#ifdef PPB_EPn
    if (ep_buf[endpoint].flags & EP_TX_PPBI /*odd*/)
        return ep_buf[endpoint].in1;
    else
        return ep_buf[endpoint].in;
#else
    return ep_buf[endpoint].in;
#endif
}
```

10.20.4.4 uint8_t usb_get_out_buffer (uint8_t *endpoint*, const unsigned char ** *buffer*)

Get a pointer to an endpoint's OUT buffer.

Call this function to get a pointer to an endpoint's OUT buffer after `usb_out_endpoint_has_data()` returns `true` (indicating that an OUT transaction has been received). Do not call this function if `usb_out_endpoint_has_data()` does not return true.

Parameters

endpoint	The endpoint requested
buffer	A pointer to a pointer which will be set to the endpoint's OUT buffer.

Returns

Return the number of bytes received.

Definition at line 1480 of file usb.c.

```
{
#ifdef PPB_EPn
    uint8_t ppbi = (ep_buf[endpoint].flags & EP_RX_PPBI)? 1: 0;

    if (ppbi /*odd*/)
        *buf = ep_buf[endpoint].out1;
    else
        *buf = ep_buf[endpoint].out;

    return BDN_LENGTH(BDSnOUT(endpoint, ppbi));
#else
    *buf = ep_buf[endpoint].out;
    return BDN_LENGTH(BDSnOUT(endpoint, 0));
#endif
}
```

10.20.4.5 bool usb_in_endpoint_busy (uint8_t *endpoint*)

Check whether an IN endpoint is busy.

An IN endpoint is said to be busy if there is data in its buffer and it is waiting for an IN token from the host in order to send it (or if it is in the process of sending the data).

Parameters

endpoint	The endpoint requested

Returns

Return true if the endpoint is busy, or false if it is not.

Definition at line 1465 of file usb.c.

```
1466 {
1467 #ifdef PPB_EPn
1468     uint8_t ppbi = (ep_buf[endpoint].flags & EP_TX_PPBI)? 1: 0;
1469     return BDSnIN(endpoint, ppbi).STAT.UOWN;
1470 #else
1471     return BDSnIN(endpoint,0).STAT.UOWN;
1472 #endif
1473 }
```

10.20.4.6 bool usb_in_endpoint_halted (uint8_t *endpoint*)

Check whether an endpoint is halted.

Check if an endpoint has been halted by the host. If an endpoint is halted, don't call usb_send_in_buffer().

See also

ENDPOINT_HALT_CALLBACK.

Parameters

endpoint	The endpoint requested

Returns

Return true if the endpointed is halted, or false if it is not.

Definition at line 1475 of file usb.c.

```
1476 {
1477     return ep_buf[endpoint].flags & EP_IN_HALT_FLAG;
1478 }
```

10.20.4.7 void usb_init (void)

Initialize the USB library and hardware.

Call this function at the beginning of execution. This function initializes the USB peripheral hardware and software library. After calling this funciton, the library will handle enumeration automatically when attached to a host.

Definition at line 487 of file usb.c.

```
488 {
489     uint8_t i;
490
491     /* Initialize the USB. 18.4 of PIC24FJ64GB004 datasheet */
492 #ifndef USB_FULL_PING_PONG_ONLY
493     SET_PING_PONG_MODE(PPB_MODE);
494 #endif
```

```
#if PPB_MODE != PPB_NONE
    SFR_USB_PING_PONG_RESET = 1;
    SFR_USB_PING_PONG_RESET = 0;
#endif
    SFR_USB_INTERRUPT_EN = 0x0;
    SFR_USB_EXTENDED_INTERRUPT_EN = 0x0;

    SFR_USB_EN = 1; /* enable USB module */

#ifdef USE_OTG
    SFR_OTGEN = 1;
#endif


#ifdef NEEDS_PULL
    SFR_PULL_EN = 1;  /* pull-up enable */
#endif

#ifdef HAS_ON_CHIP_XCVR_DIS
    SFR_ON_CHIP_XCVR_DIS = 0; /* on-chip transceiver Disable */
#endif

#ifdef HAS_LOW_SPEED
    SFR_FULL_SPEED_EN = 1;   /* Full-speed enable */
#endif

    CLEAR_USB_TOKEN_IF();   /* Clear 4 times to clear out USTAT FIFO */
    CLEAR_USB_TOKEN_IF();
    CLEAR_USB_TOKEN_IF();
    CLEAR_USB_TOKEN_IF();

    CLEAR_ALL_USB_IF();

#ifdef USB_USE_INTERRUPTS
    SFR_TRANSFER_IE = 1; /* USB Transfer Interrupt Enable */
    SFR_STALL_IE = 1;    /* USB Stall Interrupt Enable */
    SFR_RESET_IE = 1;    /* USB Reset Interrupt Enable */
#ifdef START_OF_FRAME_CALLBACK
    SFR_SOF_IE = 1;      /* USB Start-Of-Frame Interrupt Enable */
#endif
#endif

#ifdef USB_NEEDS_SET_BD_ADDR_REG
#ifdef __XC16__
    union WORD {
        struct {
            uint8_t lb;
            uint8_t hb;
        };
        uint16_t w;
        void *ptr;
    };
    union WORD w;
    w.ptr = bds;

    SFR_BD_ADDR_REG = w.hb;

#elif __XC32__
    union WORD {
        struct {
            uint8_t lb;
            uint8_t hb;
            uint8_t ub;
            uint8_t eb;
        };
        uint32_t w;
        void *ptr;
    };
    union WORD w;
    w.w = KVA_TO_PA(bds);

    SFR_BD_ADDR_REG1 = w.hb & 0xFE;
    SFR_BD_ADDR_REG2 = w.ub;
    SFR_BD_ADDR_REG3 = w.eb;
#endif
#endif

    /* These are the UEP/U1EP endpoint management registers. */

    /* Clear them all out. This is important because a bootloader
       could have set them to non-zero */
```

```
    memset(SFR_EP_MGMT(0), 0x0, sizeof(*SFR_EP_MGMT(0)) * 16);

    /* Set up Endpoint zero */
    SFR_EP_MGMT(0)->SFR_EP_MGMT_HANDSHAKE = 1; /* Endpoint handshaking enable */
    SFR_EP_MGMT(0)->SFR_EP_MGMT_CON_DIS = 0; /* 1=Disable control operations */
    SFR_EP_MGMT(0)->SFR_EP_MGMT_OUT_EN = 1; /* Endpoint Out Transaction Enable */
    SFR_EP_MGMT(0)->SFR_EP_MGMT_IN_EN = 1; /* Endpoint In Transaction Enable */
    SFR_EP_MGMT(0)->SFR_EP_MGMT_STALL = 0; /* Stall */

    for (i = 1; i <= NUM_ENDPOINT_NUMBERS; i++) {
        volatile SFR_EP_MGMT_TYPE *ep = SFR_EP_MGMT(i);
        ep->SFR_EP_MGMT_HANDSHAKE = 1; /* Endpoint handshaking enable */
        ep->SFR_EP_MGMT_CON_DIS = 1; /* 1=Disable control operations */
        ep->SFR_EP_MGMT_OUT_EN = 1; /* Endpoint Out Transaction Enable */
        ep->SFR_EP_MGMT_IN_EN = 1; /* Endpoint In Transaction Enable */
        ep->SFR_EP_MGMT_STALL = 0; /* Stall */
    }

    /* Reset the Address. */
    SFR_USB_ADDR = 0x0;
    addr_pending = 0;
    g_configuration = 0;
    ep0_buf.flags = 0;
    for (i = 0; i <= NUM_ENDPOINT_NUMBERS; i++) {
#ifdef PPB_EPn
        ep_buf[i].flags = 0;
#else
        ep_buf[i].flags = EP_RX_DTS;
#endif
    }

    memset(bds, 0x0, sizeof(bds));

    /* Setup endpoint 0 Output buffer descriptor.
       Input and output are from the HOST perspective. */
    BDS0OUT(0).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep0_buf.out);
    SET_BDN(BDS0OUT(0), BDNSTAT_UOWN, EP_0_LEN);

#ifdef PPB_EP0_OUT
    BDS0OUT(1).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep0_buf.out1);
    SET_BDN(BDS0OUT(1), BDNSTAT_UOWN, EP_0_LEN);
#endif

    /* Setup endpoint 0 Input buffer descriptor.
       Input and output are from the HOST perspective. */
    BDS0IN(0).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep0_buf.in);
    SET_BDN(BDS0IN(0), 0, EP_0_LEN);
#ifdef PPB_EP0_IN
    BDS0IN(1).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep0_buf.in1);
    SET_BDN(BDS0IN(1), 0, EP_0_LEN);
#endif

    for (i = 1; i <= NUM_ENDPOINT_NUMBERS; i++) {
        /* Setup endpoint 1 Output buffer descriptor.
           Input and output are from the HOST perspective. */
        BDSnOUT(i,0).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep_buf[i].out);
        SET_BDN(BDSnOUT(i,0), BDNSTAT_UOWN|BDNSTAT_DTSEN, ep_buf[i].out_len);
#ifdef PPB_EPn
        /* Initialize EVEN buffers when in ping-pong mode. */
        BDSnOUT(i,1).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep_buf[i].out1);
        SET_BDN(BDSnOUT(i,1), BDNSTAT_UOWN|BDNSTAT_DTSEN|BDNSTAT_DTS,
  ep_buf[i].out_len);
#endif
        /* Setup endpoint 1 Input buffer descriptor.
           Input and output are from the HOST perspective. */
        BDSnIN(i,0).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep_buf[i].in);
        SET_BDN(BDSnIN(i,0), 0, ep_buf[i].in_len);
#ifdef PPB_EPn
        /* Initialize EVEN buffers when in ping-pong mode. */
        BDSnIN(i,1).BDnADR = (BDNADR_TYPE) PHYS_ADDR(ep_buf[i].in1);
        SET_BDN(BDSnIN(i,1), 0, ep_buf[i].in_len);
#endif
    }

    #ifdef USB_NEEDS_POWER_ON
    SFR_USB_POWER = 1;
    #endif

#ifdef USE_OTG
    SFR_DPPULUP = 1;
#endif
```

```

    reset_ep0_data_stage();

#ifdef USB_USE_INTERRUPTS
    SFR_USB_IE = 1;     /* USB Interrupt enable */
#endif

    //UIRbits.URSTIF = 0; /* Clear USB Reset on Start */
}
```

10.20.4.8 bool usb_out_endpoint_halted (uint8_t *endpoint*)

Check whether an OUT endpoint is halted.

Check if an endpoint has been halted by the host. If an OUT endpoint is halted, the USB stack will automatically return STALL in response to any OUT tokens.

See also

ENDPOINT_HALT_CALLBACK.

Parameters

endpoint	The endpoint requested

Returns

Return true if the endpointed is halted, or false if it is not.

Definition at line 1542 of file usb.c.

```
{
    return ep_buf[endpoint].flags & EP_OUT_HALT_FLAG;
}
```

10.20.4.9 bool usb_out_endpoint_has_data (uint8_t *endpoint*)

Check whether an OUT endpoint has received data.

Check if an OUT endpoint has completed a transaction and has received data from the host. If it has, the application should call `usb_get_out_buffer()` to get the data and then call `usb_arm_out_endpoint()` to enable reception of the next transaction.

Parameters

endpoint	The endpoint requested

Returns

Return true if the endpoint has received data, false if it has not.

Definition at line 1497 of file usb.c.

```
{
#ifdef PPB_EPn
    uint8_t ppbi = (ep_buf[endpoint].flags & EP_RX_PPBI)? 1: 0;
    return !BDSnOUT(endpoint,ppbi).STAT.UOWN;
#else
    return !BDSnOUT(endpoint,0).STAT.UOWN;
#endif
}
```

10.20.4.10 void usb_send_data_stage (char * *buffer,* size_t *len,* usb_ep0_data_stage_callback *callback,* void * *context*)

Start the data stage of an IN control transfer.

Start the data stage of a control transfer for a transfer which has an IN data stage. Call this from `UNKNOWN_SETUP_REQUEST_CALLBACK` for IN control transfers which are being handled by the application. Once the transfer has completed, `callback` will be called with the `context` pointer provided. The `buffer` should be considered to be owned by the USB stack until the callback is called and should not be modified by the application until this time. Do not pass in a buffer which is on the stack. The data will automatically be split into as many transactions as necessary to complete the transfer.

See also

UNKNOWN_SETUP_REQUEST_CALLBACK

Parameters

buffer	A buffer containing the data to send. This should be a buffer capable of having an arbitrary lifetime. Do not use a stack variable for this buffer, and do not free this buffer until the callback has been called.
len	The number of bytes to send
callback	A callback function to call when the transfer completes. This parameter is mandatory. Once the callback is called, the transfer is over, and the buffer can be considered to be owned by the application again.
context	A pointer to be passed to the callback. The USB stack does not dereference this pointer.

Definition at line 1558 of file usb.c.

```
{
    /* Start sending the first block. Subsequent blocks will be sent
       when IN tokens are received on endpoint zero. */
    ep0_data_stage_callback = callback;
    ep0_data_stage_context = context;
    start_control_return(buffer, len, len);
}
```

10.20.4.11 void usb_send_in_buffer (uint8_t *endpoint,* size_t *len*)

Send an endpoint's IN buffer to the host.

Send the data in the IN buffer for the specified endpoint to the host. Since USB is a polled bus, this only queues the data for sending. It will actually be sent when the device receives an IN token for the specified endpoint. To check later whether the data has been sent, call `usb_in_endpoint_busy()`. If the endpoint is busy, a transmission is pending, but has not been actually transmitted yet.

Parameters

endpoint	The endpoint on which to send data
len	The amount of data to send

Definition at line 1423 of file usb.c.

```
{
#ifdef DEBUG
    if (endpoint == 0)
        error();
#endif
    if (g_configuration > 0 && !usb_in_endpoint_halted(endpoint)) {
        uint8_t pid;
        struct buffer_descriptor *bd;
```

```
#ifdef PPB_EPn
        uint8_t ppbi = (ep_buf[endpoint].flags & EP_TX_PPBI)? 1 : 0;

        bd = &BDSnIN(endpoint,ppbi);
        pid = (ep_buf[endpoint].flags & EP_TX_DTS)? 1 : 0;
        bd->STAT.BDnSTAT = 0;

        if (pid)
            SET_BDN(BDSnIN(endpoint,ppbi),
                BDNSTAT_UOWN|BDNSTAT_DTS|BDNSTAT_DTSEN, len);
        else
            SET_BDN(BDSnIN(endpoint,ppbi),
                BDNSTAT_UOWN|BDNSTAT_DTSEN, len);

        ep_buf[endpoint].flags ^= EP_TX_PPBI;
        ep_buf[endpoint].flags ^= EP_TX_DTS;
#else
        bd = &BDSnIN(endpoint,0);
        pid = (ep_buf[endpoint].flags & EP_TX_DTS)? 1 : 0;
        bd->STAT.BDnSTAT = 0;

        if (pid)
            SET_BDN(*bd,
                BDNSTAT_UOWN|BDNSTAT_DTS|BDNSTAT_DTSEN, len);
        else
            SET_BDN(*bd,
                BDNSTAT_UOWN|BDNSTAT_DTSEN, len);

        ep_buf[endpoint].flags ^= EP_TX_DTS;
#endif
    }
}
```

10.20.4.12 void usb_service (void)

Update the USB library and hardware.

This function services the USB peripheral's interrupts and handles all tasks related to enumeration and transfers. It is non-blocking. Whether an application should call this function depends on the `USB_USE_INTERRUPTS` #define. If `USB_USE_INTERRUPTS` is not defined, this function should be called periodically from the main application. If `USB_USE_INTERRUPTS` is defined, it should be called from interrupt context. On PIC24, this will happen automatically, as the interrupt handler is embedded in usb.c. On 8-bit PIC since the interrupt handlers are shared, this function will need to be called from the application's interrupt handler.

Definition at line 1293 of file usb.c.

```
{
    if (SFR_USB_RESET_IF) {
        /* A Reset was detected on the wire. Re-init the SIE. */
#ifdef USB_RESET_CALLBACK
        USB_RESET_CALLBACK();
#endif
        usb_init();
        CLEAR_USB_RESET_IF();
        SERIAL("USB Reset");
    }

    if (SFR_USB_STALL_IF) {
        /* On PIC24/32, EPSTALL bits must be cleared, or else the
         * stalled endpoint's opposite direction (eg: EP1 IN => EP1
         * OUT) will also stall (incorrectly). There is no way to
         * determine which endpoint generated this interrupt, so all
         * the endpoints' EPSTALL bits must be checked and cleared. */
        int i;
        for (i = 1; i <= NUM_ENDPOINT_NUMBERS; i++) {
            volatile SFR_EP_MGMT_TYPE *ep = SFR_EP_MGMT(i);
            ep->SFR_EP_MGMT_STALL = 0;
        }

        CLEAR_USB_STALL_IF();
    }
```

```
    if (SFR_USB_TOKEN_IF) {

        //struct ustat_bits ustat = *((struct ustat_bits*)&USTAT);

        if (SFR_USB_STATUS_EP == 0 && SFR_USB_STATUS_DIR == 0/*OUT*/) {
            /* An OUT or SETUP transaction has completed on
             * Endpoint 0.  Handle the data that was received.
             */
#ifdef PPB_EP0_OUT
            uint8_t pid = BDS0OUT(SFR_USB_STATUS_PPBI).STAT.PID;
#else
            uint8_t pid = BDS0OUT(0).STAT.PID;
#endif
            if (pid == PID_SETUP) {
                handle_ep0_setup();
            }
            else if (pid == PID_IN) {
                /* Nonsense condition:
                   (PID IN on SFR_USB_STATUS_DIR == OUT) */
            }
            else if (pid == PID_OUT) {
                handle_ep0_out();
            }
            else {
                /* Unsupported PID. Stall the Endpoint. */
                SERIAL("Unsupported PID. Stall.");
                stall_ep0();
            }

            reset_bd0_out();
        }
        else if (SFR_USB_STATUS_EP == 0 && SFR_USB_STATUS_DIR == 1/*1=IN*/) {
            /* An IN transaction has completed. The endpoint
             * needs to be re-loaded with the next transaction's
             * data if there is any.
             */
            handle_ep0_in();
        }
        else if (SFR_USB_STATUS_EP > 0 && SFR_USB_STATUS_EP <= NUM_ENDPOINT_NUMBERS) {
            if (SFR_USB_STATUS_DIR == 1 /*1=IN*/) {
                /* An IN transaction has completed. */
                SERIAL("IN transaction completed on non-EP0.");
                if (ep_buf[SFR_USB_STATUS_EP].flags & EP_IN_HALT_FLAG)
                    stall_ep_in(SFR_USB_STATUS_EP);
                else {
#ifdef IN_TRANSACTION_COMPLETE_CALLBACK
                    IN_TRANSACTION_COMPLETE_CALLBACK(SFR_USB_STATUS_EP);
#endif
                }
            }
            else {
                /* An OUT transaction has completed. */
                SERIAL("OUT transaction received on non-EP0");
                if (ep_buf[SFR_USB_STATUS_EP].flags & EP_OUT_HALT_FLAG)
                    stall_ep_out(SFR_USB_STATUS_EP);
                else {
#ifdef OUT_TRANSACTION_CALLBACK
                    OUT_TRANSACTION_CALLBACK(SFR_USB_STATUS_EP);
#endif
                }
            }
        }
        else {
            /* Transaction completed on an endpoint not used.
             * This should never happen. */
            SERIAL("Transaction completed for unknown endpoint");
        }

        CLEAR_USB_TOKEN_IF();
    }

    /* Check for Start-of-Frame interrupt. */
    if (SFR_USB_SOF_IF) {
#ifdef START_OF_FRAME_CALLBACK
        START_OF_FRAME_CALLBACK();
#endif
        CLEAR_USB_SOF_IF();
    }

    /* Check for USB Interrupt. */
    if (SFR_USB_IF) {
```

```
        SFR_USB_IF = 0;
    }
}
```

10.20.4.13 void usb_start_receive_ep0_data_stage (char * *buffer,* size_t *len,* usb_ep0_data_stage_callback *callback,* void * *context*)

Start the data stage of an OUT control transfer.

Start the data stage of a control transfer for a transfer which has an OUT data stage. Call this from `UNKNOWN_SETUP_REQUEST_CALLBACK` for OUT control transfers which being handled by the application. Once the transfer has completed, `callback` will be called with the `context` pointer provided. The `buffer` should be considered to be owned by the USB stack until the `callback` is called and should not be modified by the application until this time.

See also

UNKNOWN_SETUP_REQUEST_CALLBACK

Parameters

buffer	A buffer in which to place the data
len	The number of bytes to expect. This must be less than or equal to the number of bytes in the buffer, and for proper setup packets will be the wLength parameter.
callback	A callback function to call when the transfer completes. This parameter is mandatory. Once the callback is called, the transfer is over, and the buffer can be considered to be owned by the application again.
context	A pointer to be passed to the callback. The USB stack does not dereference this pointer

Definition at line 1547 of file usb.c.

```
{
    reset_ep0_data_stage();

    ep0_data_stage_callback = callback;
    ep0_data_stage_out_buffer = buffer;
    ep0_data_stage_buf_remaining = len;
    ep0_data_stage_context = context;
}
```

10.21 Descriptor Items

Items defined by the application which are involved in the enumeration of the device.

Functions

- int16_t USB_STRING_DESCRIPTOR_FUNC (uint8_t string_number, const void **ptr)

Variables

- const struct device_descriptor USB_DEVICE_DESCRIPTOR
- const struct configuration_descriptor * USB_CONFIG_DESCRIPTOR_MAP []

10.21.1 Detailed Description

Items defined by the application which are involved in the enumeration of the device.

The items listed in this section are macro names. An application needs to define these macro names in usb_config.h to whatever actual C names are used in the application for these items (typically in usb_descriptors.c).

It is required that the application #define these items in the application's `usb_config.h` so the USB stack can retrieve the Chapter 9 descriptors to send to the host.

While this sounds complex, it is not. See the example programs and their usb_descriptors.c that come with the USB stack for an example of what is required and how to easily implement it.

10.21.2 Function Documentation

10.21.2.1 int16_t USB_STRING_DESCRIPTOR_FUNC (uint8_t *string_number*, const void ** *ptr*)

String Descriptor Function

The USB stack will call this function to retrieve string descriptors from the application. This allows the flexibility for the application to read some strings (like serial numbers) from non-const locations (like EEPROM).

Parameters

string_number	The string number requested
ptr	A pointer to a pointer which should be set to the requested string descriptor by this function.

Returns

Return the length of the string descriptor in bytes or -1 if the string requested does not exist.

10.21.3 Variable Documentation

10.21.3.1 const struct configuration_descriptor* USB_CONFIG_DESCRIPTOR_MAP[ ]

Configuration Descriptor

This is an array of the device's configuration descriptors, as defined by the USB specification, chapter 9. USB_CON↩FIG_DESCRIPTOR_MAP must be defined to be the name of an array of pointers to configuration_descriptor objects,

often in the application's usb_descriptors.c. The order is not important because the `bConfigurationValue` field is used by the USB stack to determine the configuration number for each configuration descriptor. It is important that wTotalLength in each configuration descriptor be correct, as this is used by the USB stack to determine the number of bytes to use (It is recommended to use the sizeof() operator for this field).

See the example programs that come with the USB stack (specificallyl usb_descriptors.c) for a simple example of what is required.

10.21.3.2 const struct device_descriptor USB_DEVICE_DESCRIPTOR

Device Descriptor

This is the device's device descriptor as defined by the USB specification, chapter 9. `USB_DEVICE_DESCRIPTOR` must be defined in usb_config.h to be the name of the device descriptor structure, which will often be located in the application's usb_descriptors.c.

10.22 Static Callbacks

Optional static callback macros to be defined in the application's usb_config.h.

Optional static callback macros to be defined in the application's usb_config.h.

If desired, #define these callback functions in your application's `usb_config.h` to receive notification about specific events which happen during enumeration and otherwise. While these are not strictly required for all devices, they may be required depending on your device configuration.

10.23 USB CDC Class Enumerations and Descriptors

Packet structs, constants, and callback functions implementing the "Universal Serial Bus Class Definitions for Communication Devices" (commonly the USB CDC Specification), version 1.1.

Data Structures

- struct cdc_functional_descriptor_header
- struct cdc_acm_functional_descriptor
- struct cdc_union_functional_descriptor
- struct cdc_notification_header
- struct cdc_serial_state_notification
- struct cdc_line_coding

Macros

- #define CDC_DEVICE_CLASS 0x02 /∗ 4.1 ∗/
- #define CDC_COMMUNICATION_INTERFACE_CLASS 0x02 /∗ 4.2 ∗/
- #define CDC_COMMUNICATION_INTERFACE_CLASS_ACM_SUBCLASS 0x02 /∗ 4.3 ∗/
- #define CDC_DATA_INTERFACE_CLASS 0x0a /∗ 4.5 ∗/
- #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_NONE 0x0 /∗ 4.7 ∗/
- #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_VENDOR 0xff /∗ 4.7 ∗/

Enumerations

- enum CDCDescriptorTypes { DESC_CS_INTERFACE = 0x24, DESC_CS_ENDPOINT = 0x25 }
- enum CDCFunctionalDescriptorSubtypes { CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_HEADER = 0x0, CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_ACM = 0x2, CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_UNION = 0x6 }
- enum CDCACMCapabilities { CDC_ACM_CAPABILITY_COMM_FEATURES = 0x1, CDC_ACM_CAPABILITY_LINE_CODINGS = 0x2, CDC_ACM_CAPABILITY_SEND_BREAK = 0x4, CDC_ACM_CAPABILITY_NETWORK_CONNECTION = 0x8 }
- enum CDCRequests { CDC_SEND_ENCAPSULATED_COMMAND = 0x0, CDC_GET_ENCAPSULATED_RESPONSE = 0x1, CDC_SET_COMM_FEATURE = 0x2, CDC_GET_COMM_FEATURE = 0x3, CDC_CLEAR_COMM_FEATURE = 0x4, CDC_SET_LINE_CODING = 0x20, CDC_GET_LINE_CODING = 0x21, CDC_SET_CONTROL_LINE_STATE = 0x22, CDC_SEND_BREAK = 0x23 }
- enum CDCCommFeatureSelector { CDC_FEATURE_ABSTRACT_STATE = 0x1, CDC_FEATURE_COUNTRY_SETTING = 0x2 }
- enum CDCCharFormat { CDC_CHAR_FORMAT_1_STOP_BIT = 0, CDC_CHAR_FORMAT_1_POINT_5_STOP_BITS = 1, CDC_CHAR_FORMAT_2_STOP_BITS = 2 }
- enum CDCParityType { CDC_PARITY_NONE = 0, CDC_PARITY_ODD = 1, CDC_PARITY_EVEN = 2, CDC_PARITY_MARK = 3, CDC_PARITY_SPACE = 4 }
- enum CDCNotifications { CDC_NETWORK_CONNECTION = 0x0, CDC_RESPONSE_AVAILABLE = 0x1, CDC_SERIAL_STATE = 0x20 }

Functions

- uint8_t process_cdc_setup_request (const struct setup_packet *setup)
- int8_t CDC_SEND_ENCAPSULATED_COMMAND_CALLBACK (uint8_t interface, uint16_t length)

10.23.1 Detailed Description

Packet structs, constants, and callback functions implementing the "Universal Serial Bus Class Definitions for Communication Devices" (commonly the USB CDC Specification), version 1.1.

For more information, see the above referenced document, available from http://www.usb.org .

10.23.2 Macro Definition Documentation

10.23.2.1 #define CDC_COMMUNICATION_INTERFACE_CLASS 0x02 /* 4.2 */

Definition at line 62 of file usb_cdc.h.

10.23.2.2 #define CDC_COMMUNICATION_INTERFACE_CLASS_ACM_SUBCLASS 0x02 /* 4.3 */

Definition at line 63 of file usb_cdc.h.

10.23.2.3 #define CDC_DATA_INTERFACE_CLASS 0x0a /* 4.5 */

Definition at line 67 of file usb_cdc.h.

10.23.2.4 #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_NONE 0x0 /* 4.7 */

Definition at line 68 of file usb_cdc.h.

10.23.2.5 #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_VENDOR 0xff /* 4.7 */

Definition at line 69 of file usb_cdc.h.

10.23.2.6 #define CDC_DEVICE_CLASS 0x02 /* 4.1 */

Definition at line 61 of file usb_cdc.h.

10.23.3 Enumeration Type Documentation

10.23.3.1 enum CDCACMCapabilities

Abstract Control Management (ACM) capabilities

See section 5.2.3.3 of the CDC Specification, version 1.1.

Enumerator

CDC_ACM_CAPABILITY_COMM_FEATURES

CDC_ACM_CAPABILITY_LINE_CODINGS

CDC_ACM_CAPABILITY_SEND_BREAK

CDC_ACM_CAPABILITY_NETWORK_CONNECTION

Definition at line 92 of file usb_cdc.h.

```
92                          {
93     CDC_ACM_CAPABILITY_COMM_FEATURES = 0x1,
94     CDC_ACM_CAPABILITY_LINE_CODINGS = 0x2,
95     CDC_ACM_CAPABILITY_SEND_BREAK = 0x4,
96     CDC_ACM_CAPABILITY_NETWORK_CONNECTION = 0x8,
97 };
```

10.23.3.2 enum CDCCharFormat

CDC Character Format

These values are used in the bCharFormat field of the GET_LINE_CODING and SET_LINE_CODING requests. See section 6.2.13 (table 50) of the CDC Specification, version 1.1.

Enumerator

CDC_CHAR_FORMAT_1_STOP_BIT

CDC_CHAR_FORMAT_1_POINT_5_STOP_BITS

CDC_CHAR_FORMAT_2_STOP_BITS

Definition at line 132 of file usb_cdc.h.

```
132                     {
133     CDC_CHAR_FORMAT_1_STOP_BIT = 0,
134     CDC_CHAR_FORMAT_1_POINT_5_STOP_BITS = 1,
135     CDC_CHAR_FORMAT_2_STOP_BITS = 2,
136 };
```

10.23.3.3 enum CDCCommFeatureSelector

CDC Communication Feature Selector Codes

See section 6.2.4, Table 47 of the CDC Specification, version 1.1.

Enumerator

CDC_FEATURE_ABSTRACT_STATE

CDC_FEATURE_COUNTRY_SETTING

Definition at line 121 of file usb_cdc.h.

```
121                             {
122     CDC_FEATURE_ABSTRACT_STATE = 0x1,
123     CDC_FEATURE_COUNTRY_SETTING = 0x2,
124 };
```

10.23.3.4 enum CDCDescriptorTypes

CDC Descriptor types: 5.2.3

Enumerator

DESC_CS_INTERFACE

DESC_CS_ENDPOINT

Definition at line 74 of file usb_cdc.h.

```
74                          {
75     DESC_CS_INTERFACE = 0x24,
76     DESC_CS_ENDPOINT  = 0x25,
77 };
```

10.23.3.5 enum CDCFunctionalDescriptorSubtypes

Enumerator

CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_HEADER

CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_ACM

CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_UNION

Definition at line 80 of file usb_cdc.h.

```
80                                      {
81     CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_HEADER = 0x0,
82     CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_ACM = 0x2,
83     CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_UNION = 0x6,
84 };
```

10.23.3.6 enum CDCNotifications

CDC Class-Specific Notification Codes

See section 6.3 (table 68) of the CDC Specification, version 1.1.

Enumerator

CDC_NETWORK_CONNECTION

CDC_RESPONSE_AVAILABLE

CDC_SERIAL_STATE

Definition at line 156 of file usb_cdc.h.

```
156                         {
157     CDC_NETWORK_CONNECTION = 0x0,
158     CDC_RESPONSE_AVAILABLE = 0x1,
159     CDC_SERIAL_STATE = 0x20,
160 };
```

10.23.3.7 enum CDCParityType

CDC Parity Type

These values are used in the bParityType field of the GET_LINE_CODING and SET_LINE_CODING requests. See section 6.2.13 (table 50) of the CDC Specification, version 1.1.

Enumerator

CDC_PARITY_NONE

CDC_PARITY_ODD

CDC_PARITY_EVEN

CDC_PARITY_MARK

CDC_PARITY_SPACE

Definition at line 144 of file usb_cdc.h.

```
144                    {
145     CDC_PARITY_NONE  = 0,
146     CDC_PARITY_ODD   = 1,
147     CDC_PARITY_EVEN  = 2,
148     CDC_PARITY_MARK  = 3,
149     CDC_PARITY_SPACE = 4,
150 };
```

10.23.3.8 enum CDCRequests

CDC ACM Class Requests

These are the class requests needed for ACM (see section 6.2, table 45). Others are omitted. Get in contact with Signal 11 if you need something specific.

Enumerator

CDC_SEND_ENCAPSULATED_COMMAND

CDC_GET_ENCAPSULATED_RESPONSE

CDC_SET_COMM_FEATURE

CDC_GET_COMM_FEATURE

CDC_CLEAR_COMM_FEATURE

CDC_SET_LINE_CODING

CDC_GET_LINE_CODING

CDC_SET_CONTROL_LINE_STATE

CDC_SEND_BREAK

Definition at line 105 of file usb_cdc.h.

```
105                    {
106     CDC_SEND_ENCAPSULATED_COMMAND = 0x0,
107     CDC_GET_ENCAPSULATED_RESPONSE = 0x1,
108     CDC_SET_COMM_FEATURE = 0x2,
109     CDC_GET_COMM_FEATURE = 0x3,
110     CDC_CLEAR_COMM_FEATURE = 0x4,
111     CDC_SET_LINE_CODING = 0x20,
112     CDC_GET_LINE_CODING = 0x21,
113     CDC_SET_CONTROL_LINE_STATE = 0x22,
114     CDC_SEND_BREAK = 0x23,
115 };
```

10.23.4 Function Documentation

10.23.4.1 int8_t CDC_SEND_ENCAPSULATED_COMMAND_CALLBACK (uint8_t *interface*, uint16_t *length*)

CDC SEND_ENCAPSULATED_COMMAND callback

The USB Stack will call this function when a GET_ENCAPSULATED_COMMAND request has been received from the host. There are two ways to handle this:

1. If the request can't be handled, return -1. This will send a STALL to the host.
2. If the request can be handled, call `usb_start_receive_ep0_data_stage()` with a buffer to be filled with the command data and a callback which will get called when the data stage is complete. The callback is required, and the command data buffer passed to `usb_start_receive_ep0_data_stage()` must remain valid until the callback is called.

It is worth noting that only one control transfer can be active at any given time. Once HID_SET_REPORT_CALLBACK() has been called, it will not be called again until the next transfer, meaning that if the application-provided HID_SET_REPORT_CALLBACK() function performs option 1 above, the callback function passed to `usb_start_receive_ep0_data_stage()` will be called before any other setup transfer can happen again. Thus, it is safe to use the same buffer for all control transfers if desired.

Parameters

interface	The interface for which the command is intended
length	The length of the command which will be present in the data stage.

Returns

Return 0 if the request can be handled or -1 if it cannot. Returning -1 will cause STALL to be returned to the host.

10.23.4.2 uint8_t process_cdc_setup_request (const struct setup_packet * *setup*)

Process CDC Setup Request

Process a setup request which has been unhandled as if it is potentially a CDC setup request. This function will then call appropriate callbacks into the appliction if the setup packet is one recognized by the CDC specification.

Parameters

setup	A setup packet to handle

Returns

Returns 0 if the setup packet could be processed or -1 if it could not.

Definition at line 120 of file usb_cdc.c.

```
{
    /* The following comes from the CDC spec 1.1, chapter 6. */

    uint8_t interface = setup->wIndex;

#ifdef MULTI_CLASS_DEVICE
    /* Check the interface first to make sure the destination is a
     * CDC interface. Multi-class devices will need to call
     * cdc_set_interface_list() first.
     */
```

```
    if (!interface_is_cdc(interface))
        return -1;
#endif

#ifdef CDC_SEND_ENCAPSULATED_COMMAND_CALLBACK
    if (setup->bRequest == CDC_SEND_ENCAPSULATED_COMMAND &&
        setup->REQUEST.bmRequestType == 0x21) {
        int8_t res;
        res = CDC_SEND_ENCAPSULATED_COMMAND_CALLBACK(interface,
                                                     setup->wLength);
        if (res < 0)
            return -1;
        return 0;
    }
#endif

#ifdef CDC_GET_ENCAPSULATED_RESPONSE_CALLBACK
    if (setup->bRequest == CDC_GET_ENCAPSULATED_RESPONSE &&
        setup->REQUEST.bmRequestType == 0xa1) {
        const void *response;
        int16_t len;
        usb_ep0_data_stage_callback callback;
        void *context;

        len = CDC_GET_ENCAPSULATED_RESPONSE_CALLBACK(
                                    interface, setup->wLength,
                                    &response, &callback,
                                    &context);
        if (len < 0)
            return -1;

        usb_send_data_stage((void*)response,
                            min(len, setup->wLength),
                            callback, context);
        return 0;
    }
#endif

#ifdef CDC_SET_COMM_FEATURE_CALLBACK
    if (setup->bRequest == CDC_SET_COMM_FEATURE &&
        setup->REQUEST.bmRequestType == 0x21) {

        /* Only ABSTRACT_STATE feature is supported. If you need
         * something else here, get in contact with Signal 11. */
        if (setup->wValue != CDC_FEATURE_ABSTRACT_STATE)
            return -1;

        transfer_interface = interface;
        set_or_clear_request = setup->bRequest;
        usb_start_receive_ep0_data_stage((char*) &transfer_data.comm_feature,
                                         sizeof(transfer_data.comm_feature),
                                         set_or_clear_comm_feature_callback,
                                         NULL);
        return 0;
    }
#endif

#ifdef CDC_CLEAR_COMM_FEATURE_CALLBACK
    if (setup->bRequest == CDC_CLEAR_COMM_FEATURE &&
        setup->REQUEST.bmRequestType == 0x21) {

        /* Only ABSTRACT_STATE feature is supported. If you need
         * something else here, get in contact with Signal 11. */
        if (setup->wValue != CDC_FEATURE_ABSTRACT_STATE)
            return -1;

        transfer_interface = interface;
        set_or_clear_request = setup->bRequest;
        usb_start_receive_ep0_data_stage((char*)&transfer_data.comm_feature,
                                         sizeof(transfer_data.comm_feature),
                                         set_or_clear_comm_feature_callback,
                                         NULL);
        return 0;
    }
#endif

#ifdef CDC_GET_COMM_FEATURE_CALLBACK
    if (setup->bRequest == CDC_GET_COMM_FEATURE &&
        setup->REQUEST.bmRequestType == 0xa1) {
```

```
        bool idle_setting;
        bool data_multiplexed_state;
        int8_t res;

        /* Only ABSTRACT_STATE feature is supported. If you need
         * something else here, get in contact with Signal 11. */
        if (setup->wValue != CDC_FEATURE_ABSTRACT_STATE)
            return -1;

        res = CDC_GET_COMM_FEATURE_CALLBACK(
                                        interface,
                                        &idle_setting,
                                        &data_multiplexed_state);
        if (res < 0)
            return -1;

        transfer_data.comm_feature =
            (uint16_t) idle_setting |
                (uint16_t) data_multiplexed_state << 1;

        usb_send_data_stage((char*)&transfer_data.comm_feature,
                            min(setup->wLength,
                                sizeof(transfer_data.comm_feature)),
                            NULL/*callback*/, NULL);
        return 0;
    }
#endif

#ifdef CDC_SET_LINE_CODING_CALLBACK
    if (setup->bRequest == CDC_SET_LINE_CODING &&
        setup->REQUEST.bmRequestType == 0x21) {

        transfer_interface = interface;
        usb_start_receive_ep0_data_stage(
                              (char*)&transfer_data.line_coding,
                              min(setup->wLength,
                                  sizeof(transfer_data.line_coding)),
                              set_line_coding, NULL);
        return 0;
    }
#endif

#ifdef CDC_GET_LINE_CODING_CALLBACK
    if (setup->bRequest == CDC_GET_LINE_CODING &&
        setup->REQUEST.bmRequestType == 0xa1) {
        int8_t res;

        res = CDC_GET_LINE_CODING_CALLBACK(
                                        interface,
                                        &transfer_data.line_coding);
        if (res < 0)
            return -1;

        usb_send_data_stage((char*)&transfer_data.line_coding,
                            min(setup->wLength,
                                sizeof(transfer_data.line_coding)),
                            /*callback*/NULL, NULL);
        return 0;
    }
#endif

#ifdef CDC_SET_CONTROL_LINE_STATE_CALLBACK
    if (setup->bRequest == CDC_SET_CONTROL_LINE_STATE &&
        setup->REQUEST.bmRequestType == 0x21) {
        int8_t res;
        bool dtr = (setup->wValue & 0x1) != 0;
        bool rts = (setup->wValue & 0x2) != 0;

        res = CDC_SET_CONTROL_LINE_STATE_CALLBACK(interface, dtr, rts);
        if (res < 0)
            return -1;

        /* Return zero-length packet. No data stage. */
        usb_send_data_stage(NULL, 0, NULL, NULL);

        return 0;
    }
#endif

#ifdef CDC_SEND_BREAK_CALLBACK
    if (setup->bRequest == CDC_SEND_BREAK &&
```

```
        setup->REQUEST.bmRequestType == 0x21) {
        int8_t res;

        res = CDC_SEND_BREAK_CALLBACK(interface,
                                      setup->wValue /*duration*/);
        if (res < 0)
            return -1;

        /* Return zero-length packet. No data stage. */
        usb_send_data_stage(NULL, 0, NULL, NULL);

        return 0;
    }
#endif

    return -1;
}
```

Chapter 11

Data Structure Documentation

11.1 cdc_acm_functional_descriptor Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- uint8_t bFunctionLength
- uint8_t bDescriptorType
- uint8_t bDescriptorSubtype
- uint8_t bmCapabilities

11.1.1 Detailed Description

CDC Abstract Control Management Functional Descriptor

See Section 5.2.3.3 of the CDC Specification, version 1.1.

Definition at line 179 of file usb_cdc.h.

11.1.2 Field Documentation

11.1.2.1 uint8_t bDescriptorSubtype

CDC_DESCRIPTOR_SUBTYPE_ACM

Definition at line 182 of file usb_cdc.h.

11.1.2.2 uint8_t bDescriptorType

Use DESC_CS_INTERFACE

Definition at line 181 of file usb_cdc.h.

11.1.2.3 uint8_t bFunctionLength

Size of this functional descriptor (4)

Definition at line 180 of file usb_cdc.h.

11.1.2.4 uint8_t bmCapabilities

See CDC_ACM_CAPABILITY∗ definitions

Definition at line 183 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.2 cdc_functional_descriptor_header Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- uint8_t bFunctionLength
- uint8_t bDescriptorType
- uint8_t bDescriptorSubtype
- uint16_t bcdCDC

11.2.1 Detailed Description

CDC Functional Descriptor Header.

See section 5.2.3.1 of the CDC Specification, version 1.1.

Definition at line 168 of file usb_cdc.h.

11.2.2 Field Documentation

11.2.2.1 uint16_t bcdCDC

CDC version in BCD format. Use 0x0101 (1.1).

Definition at line 172 of file usb_cdc.h.

11.2.2.2 uint8_t bDescriptorSubtype

CDC_DESCRIPTOR_SUBTYPE_HEADER

Definition at line 171 of file usb_cdc.h.

11.2.2.3 uint8_t bDescriptorType

Use DESC_CS_INTERFACE

Definition at line 170 of file usb_cdc.h.

11.2.2.4 uint8_t bFunctionLength

Size of this functional descriptor (5)

Definition at line 169 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.3 cdc_line_coding Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- uint32_t dwDTERate
- uint8_t bCharFormat
- uint8_t bParityType
- uint8_t bDataBits

11.3.1 Detailed Description

CDC Line Coding Structure

See Section 6.2.13 of the CDC Specification, version 1.1.

Definition at line 253 of file usb_cdc.h.

11.3.2 Field Documentation

11.3.2.1 uint8_t bCharFormat

Stop bits:

See also

CDCCharFormat

Definition at line 255 of file usb_cdc.h.

11.3.2.2 uint8_t bDataBits

Data Bits: 5, 6, 7, 8 or 16

Definition at line 257 of file usb_cdc.h.

11.3.2.3 uint8_t bParityType

Parity Type:

See also

CDCParityType

Definition at line 256 of file usb_cdc.h.

11.3.2.4 uint32_t dwDTERate

Data Terminal Rate (bits per second)

Definition at line 254 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.4 cdc_notification_header Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- union {
  struct {
  uint8_t destination: 5
  uint8_t type: 2
  uint8_t direction: 1
  }
  uint8_t bmRequestType
  } REQUEST
- uint8_t bNotification
- uint16_t wValue
- uint16_t wIndex
- uint16_t wLength

11.4.1 Detailed Description

Definition at line 206 of file usb_cdc.h.

11.4.2 Field Documentation

11.4.2.1 uint8_t bmRequestType

Definition at line 213 of file usb_cdc.h.

11.4.2.2 uint8_t bNotification

See also

enum CDCNotifications

Definition at line 215 of file usb_cdc.h.

11.4.2.3 uint8_t destination

See also

enum DestinationType

Definition at line 209 of file usb_cdc.h.

11.4.2.4 uint8_t direction

0=out, 1=in

Definition at line 211 of file usb_cdc.h.

11.4.2.5 union { ... } REQUEST

11.4.2.6 uint8_t type

See also

enum RequestType

Definition at line 210 of file usb_cdc.h.

11.4.2.7 uint16_t wIndex

Definition at line 217 of file usb_cdc.h.

11.4.2.8 uint16_t wLength

Definition at line 218 of file usb_cdc.h.

11.4.2.9 uint16_t wValue

Definition at line 216 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.5 cdc_serial_state_notification Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- struct cdc_notification_header header
- union {
  struct {
    uint16_t bRxCarrier: 1
    uint16_t bTxCarrier: 1
    uint16_t bBreak: 1
    uint16_t bRingSignal: 1
    uint16_t bFraming: 1
    uint16_t bParity: 1
    uint16_t bOverrun: 1
    uint16_t __pad0__: 1
    uint16_t __pad1__: 8
  } bits
  uint16_t serial_state
} data

11.5.1 Detailed Description

Definition at line 226 of file usb_cdc.h.

11.5.2 Field Documentation

11.5.2.1 uint16_t __pad0__

Definition at line 237 of file usb_cdc.h.

11.5.2.2 uint16_t __pad1__

Definition at line 238 of file usb_cdc.h.

11.5.2.3 uint16_t bBreak

Definition at line 232 of file usb_cdc.h.

11.5.2.4 uint16_t bFraming

Definition at line 234 of file usb_cdc.h.

11.5.2.5 struct { ... } bits

11.5.2.6 uint16_t bOverrun

Definition at line 236 of file usb_cdc.h.

11.5.2.7 uint16_t bParity

Definition at line 235 of file usb_cdc.h.

11.5.2.8 uint16_t bRingSignal

Definition at line 233 of file usb_cdc.h.

11.5.2.9 uint16_t bRxCarrier

Indicates DCD

Definition at line 230 of file usb_cdc.h.

11.5.2.10 uint16_t bTxCarrier

Indicates DSR

Definition at line 231 of file usb_cdc.h.

11.5.2.11 union { ... } data

11.5.2.12 struct cdc_notification_header header

Definition at line 227 of file usb_cdc.h.

11.5.2.13 uint16_t serial_state

Definition at line 240 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.6 cdc_union_functional_descriptor Struct Reference

```
#include <usb_cdc.h>
```

Data Fields

- uint8_t bFunctionLength
- uint8_t bDescriptorType
- uint8_t bDescriptorSubtype
- uint8_t bMasterInterface
- uint8_t bSlaveInterface0

11.6.1 Detailed Description

CDC Union Functional Descriptor

See Section 5.2.3.8 of the CDC Specification, version 1.1.

Definition at line 190 of file usb_cdc.h.

11.6.2 Field Documentation

11.6.2.1 uint8_t bDescriptorSubtype

CDC_DESCRIPTOR_SUBTYPE_ACM

Definition at line 193 of file usb_cdc.h.

11.6.2.2 uint8_t bDescriptorType

Use DESC_CS_INTERFACE

Definition at line 192 of file usb_cdc.h.

11.6.2.3 uint8_t bFunctionLength

Size of this functional descriptor

Definition at line 191 of file usb_cdc.h.

11.6.2.4 uint8_t bMasterInterface

Definition at line 194 of file usb_cdc.h.

11.6.2.5 uint8_t bSlaveInterface0

Definition at line 195 of file usb_cdc.h.

The documentation for this struct was generated from the following file:

- source/usb_cdc.h

11.7 CompareBits_t Struct Reference

CompareBits_t Ring-based String Comparison Status and Control Bits

```
#include <cdm.h>
```

Data Fields

- uint8_t w

- struct {
  unsigned fMismatch: 1
  ringFrom <> ringCMP Comparison mismatch
  unsigned fWild: 1
  ringFrom <> ringCMP Comparison allows '?' wildcard
  unsigned fAlphaOnly: 1
  ringFrom <> ringCMP Comparison is SixBit Alpha Only
  };

11.7.1 Detailed Description

CompareBits_t Ring-based String Comparison Status and Control Bits

The operation of the Ring-based string comparison mechanism.

Definition at line 122 of file cdm.h.

11.7.2 Field Documentation

11.7.2.1 struct { ... }

11.7.2.2 unsigned fAlphaOnly

ringFrom <> ringCMP Comparison is SixBit Alpha Only

Definition at line 128 of file cdm.h.

11.7.2.3 unsigned fMismatch

ringFrom <> ringCMP Comparison mismatch

Definition at line 126 of file cdm.h.

11.7.2.4 unsigned fWild

ringFrom <> ringCMP Comparison allows '?' wildcard

Definition at line 127 of file cdm.h.

11.7.2.5 uint8_t w

Definition at line 124 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

11.8 ep0_buf Struct Reference

Data Fields

- unsigned char ∗const out
- unsigned char ∗const in
- unsigned char ∗const out1
- uint8_t flags

11.8.1 Detailed Description

Definition at line 313 of file usb.c.

11.8.2 Field Documentation

11.8.2.1 uint8_t flags

Definition at line 324 of file usb.c.

11.8.2.2 unsigned char∗ const in

Definition at line 315 of file usb.c.

11.8.2.3 unsigned char∗ const out

Definition at line 314 of file usb.c.

11.8.2.4 unsigned char∗ const out1

Definition at line 317 of file usb.c.

The documentation for this struct was generated from the following file:

- source/usb.c

11.9 ep_buf Struct Reference

Data Fields

- unsigned char ∗const out
- unsigned char ∗const in
- const uint8_t out_len
- const uint8_t in_len
- uint8_t flags

11.9.1 Detailed Description

Definition at line 293 of file usb.c.

11.9.2 Field Documentation

11.9.2.1 uint8_t flags

Definition at line 310 of file usb.c.

11.9.2.2 unsigned char* const in

Definition at line 295 of file usb.c.

11.9.2.3 const uint8_t in_len

Definition at line 301 of file usb.c.

11.9.2.4 unsigned char* const out

Definition at line 294 of file usb.c.

11.9.2.5 const uint8_t out_len

Definition at line 300 of file usb.c.

The documentation for this struct was generated from the following file:

- source/usb.c

11.10 ErrorBits_t Struct Reference

ErrorBits_t Ring Buffer Error indication bits

```
#include <cdm.h>
```

Data Fields

- uint16_t w
- struct {
  - unsigned fRxOver: 1
    UART Hardware Receive Overrun. Caused by slow polling.
  - unsigned fRxFraming: 1
    UART Hardware Receive Framing Error. Caused by Bad Transmission.
  - unsigned fRxParity: 1
    UART Hardware Receive Parity Error. Caused by Bad Transmission.
  - unsigned fRxRingOver: 1
    Serial Receive Ring Overrun. Probably buffer too small.
  - unsigned fTxRingOver: 1
    Serial Transmit Ring Overrun. Probably buffer too small.
  - unsigned fcdRxRingOver: 1
    PCD-Bus Receive Ring Overrun. Talker/Listener compatibility error.
  - unsigned fcdTxRingOver: 1

PCD-Bus Transmit Ring Overrun. Talker/Listener compatibility error.
unsigned fmsgRingOver: 1
Message Ring Overrun. Probably buffer too small.
unsigned fscrRingOver: 1
Scratch Ring Overrun. Probably buffer too small.
unsigned fresRingOver: 1
Result Ring Overrun. Probably buffer too small.
};

11.10.1 Detailed Description

ErrorBits_t Ring Buffer Error indication bits

These bits indicate errors associated with the various Ring Buffers.

Definition at line 138 of file cdm.h.

11.10.2 Field Documentation

11.10.2.1 struct { ... }

11.10.2.2 unsigned fcdRxRingOver

PCD-Bus Receive Ring Overrun. Talker/Listener compatibility error.

Definition at line 147 of file cdm.h.

11.10.2.3 unsigned fcdTxRingOver

PCD-Bus Transmit Ring Overrun. Talker/Listener compatibility error.

Definition at line 148 of file cdm.h.

11.10.2.4 unsigned fmsgRingOver

Message Ring Overrun. Probably buffer too small.

Definition at line 149 of file cdm.h.

11.10.2.5 unsigned fresRingOver

Result Ring Overrun. Probably buffer too small.

Definition at line 151 of file cdm.h.

11.10.2.6 unsigned fRxFraming

UART Hardware Receive Framing Error. Caused by Bad Transmission.

Definition at line 143 of file cdm.h.

11.10.2.7 unsigned fRxOver

UART Hardware Receive Overrun. Caused by slow polling.

Definition at line 142 of file cdm.h.

11.10.2.8 unsigned fRxParity

UART Hardware Receive Parity Error. Caused by Bad Transmission.

Definition at line 144 of file cdm.h.

11.10.2.9 unsigned fRxRingOver

Serial Receive Ring Overrun. Probably buffer too small.

Definition at line 145 of file cdm.h.

11.10.2.10 unsigned fscrRingOver

Scratch Ring Overrun. Probably buffer too small.

Definition at line 150 of file cdm.h.

11.10.2.11 unsigned fTxRingOver

Serial Transmit Ring Overrun. Probably buffer too small.

Definition at line 146 of file cdm.h.

11.10.2.12 uint16_t w

Definition at line 140 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

11.11 FlagBits_t Struct Reference

FlagBits_t Internal Status and Control Bits

```
#include <cdm.h>
```

Data Fields

- uint16_t w
- struct {
  - unsigned fChanged: 1
    - *Display Changed so send new display.*
  - unsigned fBlinkOn: 1

Blinking Cursor is on.

unsigned fWheel: 1

Are we using three buttons or a quadrature scroll wheel.

unsigned fSign: 1

Force '+' or '-' in putInt()

unsigned fZeroSupp: 1

Suppress sending leading zeroes with put()

unsigned fSixSupp: 1

Suppress sending leading '_' on SixBit Negative values.

unsigned fAllowCRLF: 1

Suppress sending consecutive CRLF with put()

unsigned fClockSet: 1

Rebuild the Local-time Clock structure.

unsigned fNeedID: 1

Prepend ID onto Response Sentence.

unsigned fcdLoopBack: 1

CD-Bus re-process all send messages.

};

11.11.1 Detailed Description

FlagBits_t Internal Status and Control Bits

These bits provide global control settings for some of the functions.

Definition at line 62 of file cdm.h.

11.11.2 Field Documentation

11.11.2.1 struct { ... }

11.11.2.2 unsigned fAllowCRLF

Suppress sending consecutive CRLF with put()

Definition at line 72 of file cdm.h.

11.11.2.3 unsigned fBlinkOn

Blinking Cursor is on.

Definition at line 67 of file cdm.h.

11.11.2.4 unsigned fcdLoopBack

CD-Bus re-process all send messages.

Definition at line 75 of file cdm.h.

11.11.2.5 unsigned fChanged

Display Changed so send new display.

Definition at line 66 of file cdm.h.

11.11.2.6 unsigned fClockSet

Rebuild the Local-time Clock structure.

Definition at line 73 of file cdm.h.

11.11.2.7 unsigned fNeedID

Prepend ID onto Response Sentence.

Definition at line 74 of file cdm.h.

11.11.2.8 unsigned fSign

Force '+' or '-' in putInt()

Definition at line 69 of file cdm.h.

11.11.2.9 unsigned fSixSupp

Suppress sending leading '_' on SixBit Negative values.

Definition at line 71 of file cdm.h.

11.11.2.10 unsigned fWheel

Are we using three buttons or a quadrature scroll wheel.

Definition at line 68 of file cdm.h.

11.11.2.11 unsigned fZeroSupp

Suppress sending leading zeroes with put()

Definition at line 70 of file cdm.h.

11.11.2.12 uint16_t w

Definition at line 64 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

11.12 ModbusBits_t Struct Reference

ModbusBits_t ModBus Configuration and Status Bits

```
#include <cdm.h>
```

Data Fields

- uint8_t w
- struct {
  unsigned fMaster: 1
    1=Master Mode, 0=Slave Mode
  unsigned fASCII: 1
    1=ModBus ASCII mode, 0=ModBus RTU Mode
  };

11.12.1 Detailed Description

ModbusBits_t ModBus Configuration and Status Bits

ModBus Configuration and Status Bits

Definition at line 107 of file cdm.h.

11.12.2 Field Documentation

11.12.2.1 struct { ... }

11.12.2.2 unsigned fASCII

1=ModBus ASCII mode, 0=ModBus RTU Mode

Definition at line 112 of file cdm.h.

11.12.2.3 unsigned fMaster

1=Master Mode, 0=Slave Mode

Definition at line 111 of file cdm.h.

11.12.2.4 uint8_t w

Definition at line 109 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

11.13 ModeBits_t Struct Reference

ModeBits_t Universal Controller Operating Mode Control

```
#include <cdm.h>
```

Data Fields

- uint16_t w
- struct {
  unsigned fMetric: 1
  Result values in metric.
  unsigned fUnits: 1
  Append units of measure to result values.
  unsigned fFormat: 1
  Format Results with extra characters (i.e. dates)
  unsigned fFieldID: 1
  Include Field ID= in display result.
  unsigned fUI: 1
  Handle User Interface (Display and Scrolling)
  unsigned fUIclock: 1
  Show RTC in the display.
  unsigned fBlink: 1
  User Interface has Blinking Cursor.
  unsigned fBlock: 1
  User Interface has Block Cursor.
  unsigned fUnder: 1
  User Interface has Underscore Cursor.
  unsigned fLCD: 1
  User Interface is a LCD Display Module.
  unsigned fUSB: 1
  Handle USB Hardware.
  unsigned fPollLED: 1
  LED update is polled. Default is ISR.
  unsigned fDiagLED: 1
  LEDs are diagnostic only. Non-Production.
  };

11.13.1 Detailed Description

ModeBits_t Universal Controller Operating Mode Control

These bits control the operation of various interface components.

Of primary importance is the ability to control the formatting of data values in the user display (if any) and the data reported from the sensors.

The current operating mode is maintained in EEPROM at location nvMode.

Definition at line 315 of file cdm.h.

11.13.2 Field Documentation

11.13.2.1 struct { ... }

11.13.2.2 unsigned fBlink

User Interface has Blinking Cursor.

Definition at line 326 of file cdm.h.

11.13.2.3 unsigned fBlock

User Interface has Block Cursor.

Definition at line 327 of file cdm.h.

11.13.2.4 unsigned fDiagLED

LEDs are diagnostic only. Non-Production.

Definition at line 334 of file cdm.h.

11.13.2.5 unsigned fFieldID

Include Field ID= in display result.

Definition at line 322 of file cdm.h.

11.13.2.6 unsigned fFormat

Format Results with extra characters (i.e. dates)

Definition at line 321 of file cdm.h.

11.13.2.7 unsigned fLCD

User Interface is a LCD Display Module.

Definition at line 330 of file cdm.h.

11.13.2.8 unsigned fMetric

Result values in metric.

Definition at line 319 of file cdm.h.

11.13.2.9 unsigned fPollLED

LED update is polled. Default is ISR.

Definition at line 332 of file cdm.h.

11.13.2.10 unsigned fUI

Handle User Interface (Display and Scrolling)

Definition at line 324 of file cdm.h.

11.13.2.11 unsigned fUIclock

Show RTC in the display.

Definition at line 325 of file cdm.h.

11.13.2.12 unsigned fUnder

User Interface has Underscore Cursor.

Definition at line 329 of file cdm.h.

11.13.2.13 unsigned fUnits

Append units of measure to result values.

Definition at line 320 of file cdm.h.

11.13.2.14 unsigned fUSB

Handle USB Hardware.

Definition at line 331 of file cdm.h.

11.13.2.15 uint16_t w

Definition at line 317 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

11.14 SerialBits_t Struct Reference

SerialBits_t USART Control Bits

```
#include <cdm.h>
```

Data Fields

- uint16_t w
- struct {
  - unsigned baud: 3
    Baud Rate Selector.
  - unsigned fEchoRx: 1
    Echo characters received on RX back to TX.
  - unsigned fParity: 1
    Characters have a parity bit.
  - unsigned fEven: 1
    Expect and generate Even Parity.
  - unsigned fSevenBit: 1
    Seven or eight data bits.
  - unsigned f485: 1
    RS-485 Transmit Driver Enable during TX.
  - unsigned fInhibitTX: 1
    Inhibit output from TX ring.
  - unsigned fFlow: 1
    Hardware Flow Control.

unsigned fTimeouts: 1
Maintain microsecond-accurate TX and RX times.
};

11.14.1 Detailed Description

SerialBits_t USART Control Bits

These bits set the operating mode for the Hardware USART

Definition at line 85 of file cdm.h.

11.14.2 Field Documentation

11.14.2.1 struct { ... }

11.14.2.2 unsigned baud

Baud Rate Selector.

Definition at line 89 of file cdm.h.

11.14.2.3 unsigned f485

RS-485 Transmit Driver Enable during TX.

Definition at line 94 of file cdm.h.

11.14.2.4 unsigned fEchoRx

Echo characters received on RX back to TX.

Definition at line 90 of file cdm.h.

11.14.2.5 unsigned fEven

Expect and generate Even Parity.

Definition at line 92 of file cdm.h.

11.14.2.6 unsigned fFlow

Hardware Flow Control.

Definition at line 96 of file cdm.h.

11.14.2.7 unsigned fInhibitTX

Inhibit output from TX ring.

Definition at line 95 of file cdm.h.

11.14.2.8 unsigned fParity

Characters have a parity bit.

Definition at line 91 of file cdm.h.

11.14.2.9 unsigned fSevenBit

Seven or eight data bits.

Definition at line 93 of file cdm.h.

11.14.2.10 unsigned fTimeouts

Maintain microsecond-accurate TX and RX times.

Definition at line 97 of file cdm.h.

11.14.2.11 uint16_t w

Definition at line 87 of file cdm.h.

The documentation for this struct was generated from the following file:

- source/cdm.h

Chapter 12

File Documentation

12.1 source/cdm.c File Reference

```
#include <xc.h>
#include <stdlib.h>
#include "cdm.h"
```

Functions

- uint16_t pos (const uint8_t *str)

  pos *Scan the selected ringFrom for the string, return the position*
- uint8_t scan (const uint8_t *str)

  scan *Flush the selected ringFrom up through the string*
- uint8_t scanCopy (const uint8_t *str)

  scanCopy *Copy ringFrom to ringTo up to but not including string*
- uint16_t iPeekOffset (int16_t n, uint8_t getPtr, uint8_t used, uint8_t size)

  peekOffset *Compute the offset of an indexed charactter from beginning or end of a ring*
- uint16_t peek (int16_t n)

  peek *Peek at character n in ringFrom counting from ring beginning or end*
- uint16_t get (void)

  get *Read and remove the next character from ringFrom (if any).*
- uint16_t put (uint16_t ch)

  put *Put the character ch into the Ring ringTo and handle leading zeroes and CR/LF*
- void ringReset (ringSelect_t ring)

  ringReset *Reset the pointers for the specified ring buffer, leaving it empty*
- void ringCopy (ringSelect_t ringT, ringSelect_t ringF)

  ringCopy *Copy the contents ringFrom to ringTo*
- void peekCopy (ringSelect_t ringT, ringSelect_t ringF)

  peekCopy *Copy the contents of ringFrom to ringTo without modifying ringFrom*
- void copyFromEEs (uint8_t addr, uint8_t max)

  copyFromEEs *Copy a null-terminated string from EEPROM to a Ring Buffer*
- void ensureBaud (allowedBaud_t baud)

  ensureBaud *Safely ensure that the correct baud rate divisor is set in hardware*

- uint16_t scrGet (void)

  scrGet *Read and remove the next character from scrRing (if any)*
- uint16_t scrPeek (int16_t n)

  scrPeek *Read character n counting from the beginning or end of scrRing*
- uint16_t scrPut (uint16_t ch)

  scrPut *Add the character ch to the end of scrRing*
- uint16_t msgGet (void)

  msgGet *Read and remove the next character from msgRing (if any)*
- uint16_t msgPeek (int16_t n)

  msgPeek *Read character n counting from the beginning or end of msgRing*
- uint16_t msgPut (uint16_t ch)

  msgPut *Add the character ch to the end of msgRing*
- uint16_t resGet (void)

  resGet *Read and remove the next character from resRing (if any)*
- uint16_t resPeek (int16_t n)

  resPeek *Read character n counting from the beginning or end of resRing*
- uint16_t resPut (uint16_t ch)

  resPut *Add the character ch to the end of resRing*
- uint8_t isAlpha (uint16_t ch)

  isAlpha *If the character ch is in the SixBit alphanumeric set return true*
- uint16_t cmpPut (uint16_t ch)

  cmpPut *Compare ch with the next character in ringFrom and modify fMismatch*
- void rxPoll (void)

  rxPoll *Poll the UART Hardware receiver and get chars into rxRing*
- uint16_t rxGet (void)

  rxGet *Read and remove the next character from rxRing (if any).*
- uint16_t rxPeek (int16_t n)

  rxPeek *Read character n counting from the beginning or end of rxRing*
- uint16_t txPut (uint16_t ch)

  txPut *Add a character to the Serial Port Transmit Ring Buffer txRing*
- void txPoll (void)

  txPoll *Poll the UART Hardware Transmitter and send from txRing*
- void putCRLF (void)

  putCRLF *Conditionally send a CR/LF pair to ringTo*
- void putS (const uint8_t *str)

  putS *Put Null-terminated String to ringTo*
- uint16_t putHex (uint16_t ch)

  putHex *Two hex characters or nothing if EOF*
- void putHex32 (uint32_t v)

  putHex32 *Hex value with leading zeroes suppressed*
- void putInt (int32_t v, uint8_t d)

  putInt *Signed integer with optional leading zeroes to ringTo*
- uint8_t sixNybble (uint8_t v)

  sixNybble *Convert six bits into a printable sixBit character*
- void putSix (uint32_t v)

  putSix *Integer Output in Sixbit without leading '0' or '_'*
- uint32_t getSix (void)

getSix *Get variable-length Sixbit value from selected ring*

- uint16_t hexNybble (uint16_t ch)

  hexNybble *Convert hexadecimal nybble value*

- uint32_t getHex (void)

  getHex *Get variable length hexadecimal value from ringFrom*

- int32_t getInt (void)

  getInt *Get decimal integer from ringFrom*

- uint8_t eeGet (uint8_t addr)

  eeGet *Set the EEADR and Read character from EEPROM*

- uint8_t eePeek (uint8_t n)

  eePeek *Read character from EEPROM relative to the current EEADR*

- uint16_t eePut (uint16_t ch)

  eePut *Write the character ch to the current EEADR and increment the address*

- uint16_t eeGet16 (uint8_t addr)

  eeGet16 *Get an unsigned 16-bit value from EEPROM*

- uint32_t eeGet32 (uint8_t addr)

  eeGet32 *Get an unsigned 32-bit value from EEPROM*

- void eePut8 (uint16_t val)

  eePut8 *Write the byte value to the current EEADR and increment the address*

- void eePut16 (uint16_t val)

  eePut16 *Write the unsigned 16-bit value to the current EEADR and increment the address*

- void eePut32 (uint32_t val)

  eePut32 *Write the unsigned 32-bit value to the current EEADR and increment the address*

- uint16_t putDumpChar (uint16_t ch)

  putDumpChar *Send the given character to the ringTo Ring Buffer as a printable form*

- uint16_t putCtrlChar (uint16_t ch)

  putCtrlChar *Send the given character to the ringTo Ring Buffer as a printable form*

- void eeDump (uint8_t addr, uint8_t n)

  eeDump *Dump a row of bytes from EEPROM to the Ring Buffer ringTo.*

- void eeDumpAll (void)

  eeDumpAll *Make a user-friendly dump of all of EEPROM to ringTo*

- void sleepLevel (uint8_t level)

  sleepLevel *Select Power level: 0=sleep, 1=running, 2=Aux On*

- uint8_t parity (uint8_t ch)

  parity *Compute the parity of the character ch*

- uint16_t checkSum (ringSelect_t ring)

  checkSum *Verify a NMEA-style Checksum of bytes in ring*

- uint16_t modbusCRC16 ()

  modbusCRC16 *Compute the Modbus CRC-16 for the contents of ringFrom*

- uint32_t modbusTimeout (modbusTimeoutSelector_t selector)

  modbusTimeout *Return the selected timeout value based on current baud rate*

- uint16_t diPut (uint16_t ch)

  diPut *Put a character in the LCD Display at locction diCursor*

- void diPoll (void)

  diPoll *Send contents of diDisplay buffer to LCD Display as needed*

- void diBlankField (uint8_t n)

  diBlankField *Blank to the end of the field ending at position n*

- void uiPoll (void)

  uiPoll *Poll the buttons on the user interface to detect bounce and held-down*
- void adcPoll (void)

  adcPoll *Poll the enabled A/D converter channels and convert readings*
- void adcDisable (void)

  adcDisable *Disable all ADC operation for Sleep Mode*
- void adcShowChannel (uint8_t channel)

  adcShowChannel *Show the formatted value of the selected Analog channel*
- void adcShow (uint8_t channel)

  adcShow *Show the current value of the selected ADC channel or all enabled channels*
- void rtcTick (void)

  rtcTick *Once-per-second update of the Real Time Clock, Calendar and elapsed timers*
- void rtcSet (uint8_t y, uint8_t m, uint8_t d)

  rtcSet *Set the Local Time calendar in the Real Time Clock*
- void rtcGetSavedParams (void)

  rtcGetSavedParams *Validate RTC EEPROM info and Total Running Time*
- void rtcISR (void)

  rtcISR *8-per-second interrupt for clock and LED blinking*
- uint32_t rtcMicroSeconds ()

  rtcMicroSeconds *Returns the current timer value in pseudo-microseconds.*
- uint8_t rtcPoll (void)

  rtcPoll *Clock Polling handles the rather infrequent operations such as calendar updates*
- void rtcDateTime (uint8_t format)

  rtcDateTime *Make a display of the date and time controlled by select bits in the parameter*
- void pollCritical (void)

  pollCritical *Poll the time-critical hardware*
- uint8_t wait (uint8_t secs)

  wait *Wait the specified number of seconds while ensuring that all polling is handled*
- uint16_t cdGet (void)

  cdGet *Get the next character from the PCD-Bus Input message packet*
- uint16_t cdPeek (int16_t n)

  cdPeek *Peek at a character in the PCD-Bus Input message packet*
- uint16_t cdPut (uint16_t ch)

  cdPut *Add a character to the PCD-Bus Output message packet*
- void cdBlindSend (void)

  cdBlindSend *Send the contents of cdTxRing to the bus, discarding any Rx messages*
- uint16_t crc8 (uint16_t iCRC, uint8_t b)

  crc8 *Compute updated value of CRC8 register for a given byte.*
- void loopDelay (int16_t k)

  loopDelay *Insert a delay to compensate for hardware issued like cable length and capacitance*
- void cdPoll (void)

  cdPoll *Poll the PCD-Bus waiting for the start of a message packet*
- void showVersion (uint8_t n)

  showVersion *Show Version and Version Date as selected by parameter bits*
- void getPhrase (ringSelect_t ringT, ringSelect_t ringF)

  getPhrase *Copy the next delimited command phrase from ringFrom to the scrRing*
- void putFieldDelimiter (void)

putFieldDelimiter *Put the field delimiter in the result ring with possible leading sender ID*

- void putSimpleFieldID (uint8_t ch)

  putSimpleFieldID *Output a properly formatted Field Identifier (like ',S:')*

- void putIndexedFieldID (uint8_t ch, uint16_t n)

  putIndexedFieldID *Output a properly formatted Field Identifier (like ',a3:')*

- uint8_t doPhrase (ringSelect_t ringT, ringSelect_t ringF)

  doPhrase *Process the single Command that is already isolated in ringF*

- void doSentence (ringSelect_t ringT, ringSelect_t ringF)

  doSentence *Verify the checksum and process the sequence of phrases*

- uint8_t performGSMcycle (uint8_t destN)

  performGSMcycle *Send communication sequence to the GSM radio*

- uint8_t performATOK (void)

  performATOK *Send commands to the radio modem and verify response*

- uint8_t performATCREG (uint8_t ring)

  performATCREG *Send commands to the radio modem and verify registration*

- uint8_t performATCSQ (uint8_t ring)

  performATCSQ *Send commands to the radio modem and get signal quality into ring*

- uint8_t performATCMGF (void)

  performATCMGF *Send commands to the radio modem and set Text Mode Format*

- uint8_t performATCMGS (uint8_t dest, uint8_t ring)

  performATCMGS *Send commands to the radio modem and Send Message from ring*

- uint8_t performATCMGL (uint8_t ringDN, uint8_t ringCMD)

  performATCMGL *Send commands to the radio modem and get inbound commands to ringCMD*

- uint8_t performATCMGD (uint8_t n)

  performATCMGD *Send commands to the radio modem and delete the specified SMS message*

- uint8_t performLaser (uint8_t ring)

  performLaser *Send commands to the Laser Module and format the results into ring*

- void laserOnTest (void)

  laserOnTest *Send commands to the Laser Module to leave the laser on for 30 seconds*

- void traceDump (uint8_t index)

  traceDump *Send a copy of whatever is in ringFrom to the PCD-bus as a Znn: message*

- void initMessage (void)

  initMessage *Power-On Initialization Message identifying the module*

- void Virgin (void)

  Virgin *Module has just been manufactured and has no actual assigned Style*

- void Serial (void)

  Serial *Module is the Manufacturing Serial Interface*

- void GSM (void)

  GSM *Module has a GSM Cellular Radio Modem for two-way SMS Messages*

- void Laser (void)

  Laser *Module has a Laser Distance Measurement sensor for fluid level measurement*

- void User (void)

  User *Module has a User Interface consisting of an LCD Display and buttons or a Scroll Wheel*

- void USB (void)

  USB *Module implements the on-chip USB Serial Port interface*

- void Test (void)

  Test *Module is a general-purpose test unit*

- void CDMA (void)

  CDMA *Module has a CDMA Cellular Radio Modem for two-way SMS messages*
- void RS485 (void)

  RS485 *Module has a RS485 Serial Interface*
- void Modbus (void)

  Modbus *Module is a ModBus Interface*
- void menuStep (void)

  menuStep *...*
- void menu (void)

  menu *Handle operation of the Menu state machine*
- void eeInit ()

  eeInit *Initialize the EEPROM just after manufacturing since the compiler can't do it*

Variables

- volatile unsigned char WPUB

12.1.1 Variable Documentation

12.1.1.1 volatile unsigned char WPUB

12.2 source/cdm.h File Reference

Control-Data Module Firmware.

```
#include <stdint.h>
```

Data Structures

- struct FlagBits_t

  FlagBits_t *Internal Status and Control Bits*
- struct SerialBits_t

  SerialBits_t *USART Control Bits*
- struct ModbusBits_t

  ModbusBits_t *ModBus Configuration and Status Bits*
- struct CompareBits_t

  CompareBits_t *Ring-based String Comparison Status and Control Bits*
- struct ErrorBits_t

  ErrorBits_t *Ring Buffer Error indication bits*
- struct ModeBits_t

  ModeBits_t *Universal Controller Operating Mode Control*

Macros

- #define versionNumber (0x01010105)

  Firmware Version Number.
- #define versionDate (0x20141019)

  Firmware Compilation Date.
- #define leftLED (LATAbits.LA4)

  Output bit that controls the Left LED.
- #define leftLEDtris (TRISAbits.RA4)

  TRIS bit that controls the Left LED.
- #define rightLED (LATAbits.LA5)

  Output bit that controls the Right LED.
- #define rightLEDtris (TRISAbits.RA5)

  TRIS bit that controls the Right LED.
- #define RTSout (LATCbits.LC0)

  Output bit for Modem RTS and Turns on RS-485 Drivers.
- #define RTSouttris (TRISCbits.TRISC0)

  TRIS bit should be 0 for output.
- #define CTSin (PORTCbits.RC1)

  Input bit for Modem CTS.
- #define CTSintris (TRISCbits.TRISC1)

  TRIS bit should be 1 for input.
- #define DSRin (PORTCbits.RC2)

  Input bit for Modem DSR.
- #define DSRintris (TRISCbits.TRISC2)

  TRIS bit should be 1 for input.
- #define nvVer (0x00)

  32-Bit Firmware Version Number
- #define nvDate (0x04)

  32-Bit Manufacturing Date
- #define nvSN (0x08)

  32-bit Device Serial Number
- #define nvTRT (0x0C)

  32-bit Total Running Time (periodically updated)
- #define nvRTO (0x10)

  32-bit Running Time Offset to UTC
- #define nvZone (0x14)

  16-bit Time Zone Offset in minutes
- #define nvStyle (0x16)

  8-bit Board Style selects hardware feature drivers
- #define nvAvail1 (0x17)

  8-bit unused =====
- #define nvReset (0x18)

  Count of Power-On Resets.
- #define nvSeq (0x19)

  Sequence Number of last transmitted message.
- #define nvMode (0x1A)

16-bit Operating Mode bits >> ModeBits.w

- #define nvModbus (0x1C)

  8-bit ModBus Configuration Bits >> ModbusBits.w
- #define nvSlave (0x1D)

  8-bit ModBus Slave Address (1-247 allowed)
- #define nvSerial (0x1E)

  16-bit Serial Port Configuration >> SerialBits.w
- #define nvID (0x20)

  16-byte Device ID - User assignable
- #define nvModel (0x30)

  16-byte Device Model Number/Description
- #define nvDest0 (0x40)

  Report Destination 0 (like a phone number)
- #define nvDest1 (0x50)

  Report Destination 1 (like a phone number)
- #define nvDest2 (0x60)

  Report Destination 2 (like a phone number)
- #define nvCmd0 (0x70)

  Command to run when Interval 0 expires (32 bytes)
- #define nvCmd1 (0x90)

  Command to run when Interval 1 expires (32 bytes)
- #define nvCmd2 (0xB0)

  Command to run when Interval 2 expires (32 bytes)
- #define nvInt0 (0xD0)

  Interval 0 Reload in Seconds (4 bytes)
- #define nvInt1 (0xD4)

  Interval 1 Reload in Seconds (4 bytes)
- #define nvInt2 (0xD8)

  Interval 2 Reload in Seconds (4 bytes)
- #define uiButtons (PORTC)

  PIC Port containing User Interface input bits.
- #define uiButtonMask (uiUp | uiDown | uiEnter)

  Mask for input bits in port.
- #define uiUp (0x01)

  Input bit mask for the UP button.
- #define uiEnter (0x02)

  Input bit mask for the ENTER button.
- #define uiDown (0x04)

  Input bit mask for the DOWN button.
- #define rtcReload (5859)

  Reload Value for 8/sec no prescaler 48MHz / 1024.
- #define fPower (LATAbits.LA3)

  Output bit that controls peripheral power.
- #define activeDelay {LATBbits.LB0 = 0; LATBbits.LB1 = 0; LATBbits.LB4 = 1; LATBbits.LB5 = 1; }
- #define testC (PORTBbits.RB0)
- #define testD (PORTBbits.RB1)
- #define releaseC {TRISBbits.RB0 = 1; TRISBbits.RB4 = 1; if(ModeBits.fDiagLED)rightLED = 0; }

- #define releaseD {TRISBbits.RB1 = 1; TRISBbits.RB5 = 1; if(ModeBits.fDiagLED)leftLED = 0; }
- #define setChigh {TRISBbits.RB0 = 1; TRISBbits.RB4 = 0; activeDelay; TRISBbits.RB4 = 1; if(ModeBits.fDiagL↩ ED)rightLED = 0; }
- #define setDhigh {TRISBbits.RB1 = 1; TRISBbits.RB5 = 0; activeDelay; TRISBbits.RB5 = 1; if(ModeBits.fDiagL↩ ED)leftLED = 0; }
- #define setClow {TRISBbits.RB0 = 0; activeDelay; if(ModeBits.fDiagLED)rightLED = 1; }
- #define setDlow {TRISBbits.RB1 = 0; activeDelay; if(ModeBits.fDiagLED)leftLED = 1; }

Enumerations

- enum style_t {
  styleVirgin, styleUser, styleLaser, styleCDMA,
  styleGSM, styleUSB, styleBlue, styleWiFi,
  styleModbus, styleSNAP, styleTest, styleSerial,
  styleRS485 }

  style_t *Board Configuration Style*
- enum allowedBaud_t {
  baud1200, baud2400, baud4800, baud9600,
  baud19200, baud57600, baud115200, baudSpecial }

  allowedBaud_t *Allowed Baud Rates for the UART*
- enum ringSelect_t {
  ringNULL, ringRX, ringTX, ringCDrx,
  ringCDtx, ringSCR, ringRES, ringMSG,
  ringUSB, ringEE, ringCMP, ringDisp }

  ringSelect_t *Selection of the ring Buffers for get(), peek() and put()*
- enum modbusTimeoutSelector_t { t15, t35, turnaround, response }

  modbusTimeoutSelector_t *Different Timeout values used to implement ModBus*

Functions

- uint16_t pos (const uint8_t ∗str)

  pos *Scan the selected ringFrom for the string, return the position*
- uint8_t scan (const uint8_t ∗str)

  scan *Flush the selected ringFrom up through the string*
- uint8_t scanCopy (const uint8_t ∗str)

  scanCopy *Copy ringFrom to ringTo up to but not including string*
- uint16_t put (uint16_t ch)

  put *Put the character ch into the Ring ringTo and handle leading zeroes and CR/LF*
- uint16_t get (void)

  get *Read and remove the next character from ringFrom (if any).*
- void ringReset (ringSelect_t ring)

  ringReset *Reset the pointers for the specified ring buffer, leaving it empty*
- uint16_t peek (int16_t n)

  peek *Peek at character n in ringFrom counting from ring beginning or end*
- uint16_t iPeekOffset (int16_t n, uint8_t getPtr, uint8_t used, uint8_t size)

  peekOffset *Compute the offset of an indexed charactter from beginning or end of a ring*
- void ringCopy (ringSelect_t ringT, ringSelect_t ringF)

  ringCopy *Copy the contents ringFrom to ringTo*

- void peekCopy (ringSelect_t ringT, ringSelect_t ringF)

  peekCopy *Copy the contents of ringFrom to ringTo without modifying ringFrom*
- void copyFromEEs (uint8_t addr, uint8_t max)

  copyFromEEs *Copy a null-terminated string from EEPROM to a Ring Buffer*
- uint16_t scrGet (void)

  scrGet *Read and remove the next character from scrRing (if any)*
- uint16_t scrPeek (int16_t n)

  scrPeek *Read character n counting from the beginning or end of scrRing*
- uint16_t scrPut (uint16_t ch)

  scrPut *Add the character ch to the end of scrRing*
- uint16_t msgGet (void)

  msgGet *Read and remove the next character from msgRing (if any)*
- uint16_t msgPeek (int16_t n)

  msgPeek *Read character n counting from the beginning or end of msgRing*
- uint16_t msgPut (uint16_t ch)

  msgPut *Add the character ch to the end of msgRing*
- uint16_t resGet (void)

  resGet *Read and remove the next character from resRing (if any)*
- uint16_t resPeek (int16_t n)

  resPeek *Read character n counting from the beginning or end of resRing*
- uint16_t resPut (uint16_t ch)

  resPut *Add the character ch to the end of resRing*
- uint8_t isAlpha (uint16_t ch)

  isAlpha *If the character ch is in the SixBit alphanumeric set return true*
- uint16_t cmpPut (uint16_t ch)

  cmpPut *Compare ch with the next character in ringFrom and modify fMismatch*
- uint16_t usbGet (void)

  usbGet *...*
- uint16_t usbPeek (int16_t n)

  usbPeek *...*
- uint16_t usbPut (uint16_t ch)

  usbPut *...*
- uint16_t rxGet (void)

  rxGet *Read and remove the next character from rxRing (if any).*
- uint16_t rxPeek (int16_t n)

  rxPeek *Read character n counting from the beginning or end of rxRing*
- void rxPoll (void)

  rxPoll *Poll the UART Hardware receiver and get chars into rxRing*
- uint16_t txPut (uint16_t ch)

  txPut *Add a character to the Serial Port Transmit Ring Buffer txRing*
- void txPoll (void)

  txPoll *Poll the UART Hardware Transmitter and send from txRing*
- uint16_t putDumpChar (uint16_t ch)

  putDumpChar *Send the given character to the ringTo Ring Buffer as a printable form*
- uint16_t putCtrlChar (uint16_t ch)

  putCtrlChar *Send the given character to the ringTo Ring Buffer as a printable form*
- void ensureBaud (allowedBaud_t baud)

ensureBaud *Safely ensure that the correct baud rate divisor is set in hardware*

- uint8_t parity (uint8_t ch)

  parity *Compute the parity of the character ch*

- uint16_t checkSum (ringSelect_t ring)

  checkSum *Verify a NMEA-style Checksum of bytes in ring*

- uint16_t modbusCRC16 ()

  modbusCRC16 *Compute the Modbus CRC-16 for the contents of ringFrom*

- uint16_t crc8 (uint16_t iCRC, uint8_t b)

  crc8 *Compute updated value of CRC8 register for a given byte.*

- uint32_t modbusTimeout (modbusTimeoutSelector_t selector)

  modbusTimeout *Return the selected timeout value based on current baud rate*

- void putCRLF (void)

  putCRLF *Conditionally send a CR/LF pair to ringTo*

- void putS (const uint8_t ∗str)

  putS *Put Null-terminated String to ringTo*

- uint16_t putHex (uint16_t ch)

  putHex *Two hex characters or nothing if EOF*

- void putHex32 (uint32_t v)

  putHex32 *Hex value with leading zeroes suppressed*

- void putInt (int32_t v, uint8_t d)

  putInt *Signed integer with optional leading zeroes to ringTo*

- uint8_t sixNybble (uint8_t v)

  sixNybble *Convert six bits into a printable sixBit character*

- void putSix (uint32_t v)

  putSix *Integer Output in Sixbit without leading '0' or '_'*

- uint32_t getSix (void)

  getSix *Get variable-length Sixbit value from selected ring*

- uint16_t hexNybble (uint16_t ch)

  hexNybble *Convert hexadecimal nybble value*

- uint32_t getHex (void)

  getHex *Get variable length hexadecimal value from ringFrom*

- int32_t getInt (void)

  getInt *Get decimal integer from ringFrom*

- uint8_t eeGet (uint8_t addr)

  eeGet *Set the EEADR and Read character from EEPROM*

- uint8_t eePeek (uint8_t n)

  eePeek *Read character from EEPROM relative to the current EEADR*

- uint16_t eePut (uint16_t ch)

  eePut *Write the character ch to the current EEADR and increment the address*

- uint16_t eeGet16 (uint8_t addr)

  eeGet16 *Get an unsigned 16-bit value from EEPROM*

- uint32_t eeGet32 (uint8_t addr)

  eeGet32 *Get an unsigned 32-bit value from EEPROM*

- void eePut8 (uint16_t val)

  eePut8 *Write the byte value to the current EEADR and increment the address*

- void eePut16 (uint16_t val)

  eePut16 *Write the unsigned 16-bit value to the current EEADR and increment the address*

- void eePut32 (uint32_t val)

  eePut32 *Write the unsigned 32-bit value to the current EEADR and increment the address*
- void eeDump (uint8_t addr, uint8_t n)

  eeDump *Dump a row of bytes from EEPROM to the Ring Buffer ringTo.*
- void eeDumpAll (void)

  eeDumpAll *Make a user-friendly dump of all of EEPROM to ringTo*
- uint16_t diPut (uint16_t ch)

  diPut *Put a character in the LCD Display at locction diCursor*
- void diPoll (void)

  diPoll *Send contents of diDisplay buffer to LCD Display as needed*
- void diBlankField (uint8_t n)

  diBlankField *Blank to the end of the field ending at position n*
- void uiPoll (void)

  uiPoll *Poll the buttons on the user interface to detect bounce and held-down*
- void adcPoll (void)

  adcPoll *Poll the enabled A/D converter channels and convert readings*
- void adcDisable (void)

  adcDisable *Disable all ADC operation for Sleep Mode*
- void adcShowChannel (uint8_t channel)

  adcShowChannel *Show the formatted value of the selected Analog channel*
- void adcShow (uint8_t channel)

  adcShow *Show the current value of the selected ADC channel or all enabled channels*
- void sleepLevel (uint8_t level)

  sleepLevel *Select Power level: 0=sleep, 1=running, 2=Aux On*
- void rtcTick (void)

  rtcTick *Once-per-second update of the Real Time Clock, Calendar and elapsed timers*
- void rtcSet (uint8_t y, uint8_t m, uint8_t d)

  rtcSet *Set the Local Time calendar in the Real Time Clock*
- void rtcGetSavedParams (void)

  rtcGetSavedParams *Validate RTC EEPROM info and Total Running Time*
- void rtcISR (void)

  rtcISR *8-per-second interrupt for clock and LED blinking*
- uint8_t rtcPoll (void)

  rtcPoll *Clock Polling handles the rather infrequent operations such as calendar updates*
- uint32_t rtcMicroSeconds ()

  rtcMicroSeconds *Returns the current timer value in pseudo-microseconds.*
- void rtcDateTime (uint8_t format)

  rtcDateTime *Make a display of the date and time controlled by select bits in the parameter*
- void pollCritical (void)

  pollCritical *Poll the time-critical hardware*
- uint8_t wait (uint8_t secs)

  wait *Wait the specified number of seconds while ensuring that all polling is handled*
- uint16_t cdGet (void)

  cdGet *Get the next character from the PCD-Bus Input message packet*
- uint16_t cdPeek (int16_t n)

  cdPeek *Peek at a character in the PCD-Bus Input message packet*
- uint16_t cdPut (uint16_t ch)

cdPut *Add a character to the PCD-Bus Output message packet*

- void cdBlindSend (void)

  cdBlindSend *Send the contents of cdTxRing to the bus, discarding any Rx messages*
- void loopDelay (int16_t k)

  loopDelay *Insert a delay to compensate for hardware issued like cable length and capacitance*
- void cdPoll (void)

  cdPoll *Poll the PCD-Bus waiting for the start of a message packet*
- void showVersion (uint8_t n)

  showVersion *Show Version and Version Date as selected by parameter bits*
- void getPhrase (ringSelect_t ringT, ringSelect_t ringF)

  getPhrase *Copy the next delimited command phrase from ringFrom to the scrRing*
- void putFieldDelimiter (void)

  putFieldDelimiter *Put the field delimiter in the result ring with possible leading sender ID*
- void putSimpleFieldID (uint8_t ch)

  putSimpleFieldID *Output a properly formatted Field Identifier (like ',S:')*
- void putIndexedFieldID (uint8_t ch, uint16_t n)

  putIndexedFieldID *Output a properly formatted Field Identifier (like ',a3:')*
- uint8_t doPhrase (ringSelect_t ringT, ringSelect_t ringF)

  doPhrase *Process the single Command that is already isolated in ringF*
- void doSentence (ringSelect_t ringT, ringSelect_t ringF)

  doSentence *Verify the checksum and process the sequence of phrases*
- void menuStep (void)

  menuStep *...*
- void menu (void)

  menu *Handle operation of the Menu state machine*
- void initMessage (void)

  initMessage *Power-On Initialization Message identifying the module*
- void Virgin (void)

  Virgin *Module has just been manufactured and has no actual assigned Style*
- void Serial (void)

  Serial *Module is the Manufacturing Serial Interface*
- void Test (void)

  Test *Module is a general-purpose test unit*
- void RS485 (void)

  RS485 *Module has a RS485 Serial Interface*
- void User (void)

  User *Module has a User Interface consisting of an LCD Display and buttons or a Scroll Wheel*
- void USB (void)

  USB *Module implements the on-chip USB Serial Port interface*
- void CDMA (void)

  CDMA *Module has a CDMA Cellular Radio Modem for two-way SMS messages*
- void GSM (void)

  GSM *Module has a GSM Cellular Radio Modem for two-way SMS Messages*
- uint8_t performGSMcycle (uint8_t destN)

  performGSMcycle *Send communication sequence to the GSM radio*
- uint8_t performATOK (void)

  performATOK *Send commands to the radio modem and verify response*

- uint8_t performATCREG (uint8_t ring)

  performATCREG *Send commands to the radio modem and verify registration*
- uint8_t performATCSQ (uint8_t ring)

  performATCSQ *Send commands to the radio modem and get signal quality into ring*
- uint8_t performATCMGF (void)

  performATCMGF *Send commands to the radio modem and set Text Mode Format*
- uint8_t performATCMGS (uint8_t dest, uint8_t ring)

  performATCMGS *Send commands to the radio modem and Send Message from ring*
- uint8_t performATCMGL (uint8_t ringDN, uint8_t ringCMD)

  performATCMGL *Send commands to the radio modem and get inbound commands to ringCMD*
- uint8_t performATCMGD (uint8_t n)

  performATCMGD *Send commands to the radio modem and delete the specified SMS message*
- void Laser (void)

  Laser *Module has a Laser Distance Measurement sensor for fluid level measurement*
- uint8_t performLaser (uint8_t ring)

  performLaser *Send commands to the Laser Module and format the results into ring*
- void laserOnTest (void)

  laserOnTest *Send commands to the Laser Module to leave the laser on for 30 seconds*
- void traceDump (uint8_t index)

  traceDump *Send a copy of whatever is in ringFrom to the PCD-bus as a Znn: message*
- void Modbus (void)

  Modbus *Module is a ModBus Interface*
- void eeInit ()

  eeInit *Initialize the EEPROM just after manufacturing since the compiler can't do it*

Variables

- uint32_t rxMicroSeconds

  Precision time of last received character.
- uint8_t diTicks

  Fast Tick last time the display was updated.
- uint8_t diCursor = 0xFF

  Cursor Position (0..31), or 0xFF for reset.
- uint8_t uiCursor = 0xFF

  Position of data entry cursor.
- uint8_t diDisplay [32]

  LCD Display Buffer. Size must be a power of two.
- uiState_t uiState = none
- uint8_t uiTicks

  Tick counter to time button-held-down.
- uint8_t uiCurrent

  Most recently read button bit pattern.
- uint8_t uiChanged

  Bits that just changed.
- uint8_t uiLast

  Previous button bit pattern.
- uint8_t uiHeld

Ticks since last change.

- int8_t uiScroll

  Signed running total of Up and Down counts.
- uint16_t adcResult [3]

  Current values of the results from each ADC channel.
- uint8_t adcDisplayInterval

  Display interval for ADC results, if used.
- uint32_t rtcElapsed

  Running Time in Seconds since power on.
- uint8_t rtcYear

  Local Time Calendar Year (14-99)
- uint8_t rtcMonth

  Local Time Calendar Month (1-12)
- uint8_t rtcDay

  Local Time Calendar Day (1-31)
- uint8_t rtcDOW

  Local Time Day of week (1-7, Sun-Sat)
- uint32_t rtcClock

  Local Clock Time in Seconds within current day.
- uint32_t rtcTimerA

  One of three general-purpose down-counters (in seconds)
- uint32_t rtcTimerB

  One of three general-purpose down-counters (in seconds)
- uint32_t rtcTimerC

  One of three general-purpose down-counters (in seconds)
- uint8_t rtcTicks

  Counts Fast interrupts for sub-second timing (0..7) >= 8 when a second passes.
- uint8_t ledMask

  Bit mask for LED blinking pattern.
- uint8_t ledA

  Current flashing pattern for Left LED.
- uint8_t ledB

  Current flashing pattern for Rigt LED.
- uint8_t ledAreload

  Default flashing pattern for Left LED.
- uint8_t ledBreload

  Default flashing pattern for Right LED.
- uint16_t pollTime

  Timer value at previous Critical Poll.
- uint16_t pollLongest

  Longest Critical Polling interval.
- uint8_t cdDelayCount

  Loop Delays for CD-Bus Timeouts.
- uint8_t cdDiagRD

  Received message count (discarded)
- uint8_t cdDiagRB

  Received message count.

- uint8_t cdDiagRE

  Received message Error count.
- uint8_t cdDiagTB

  Transmitted message count.
- uint8_t cdDiagTE

  Transmitted message Error count.
- cdState_t cdState = cdWaitIdle
- uint8_t lastCommandSeq
- uint32_t lastCommandSender
- uint8_t menuState

  Index for the Menu Tree State Machine.
- uint8_t opPhase

  Index for the individual Style State Machines.

- #define delimCS ('∗')

  Delimiter for Checksum.
- #define delimPhrase (';')

  Delimiter for Command Phrases.
- #define delimSubParam (',')

  Delimiter for Sub-Parameters.
- #define delimAssign ('=')

  Delimiter for Parameter Value Assignment.
- #define delimQuery ('?')

  Delimiter for Parameter Value Query.
- #define delimReport (':')

  Delimiter for Parameter Value Report.
- #define delimTest ('∼')

  Delimiter for Parameter Value Test.
- #define delimQuot1 (0x22)

  Delimiter for Quoted Strings - Double Quote '"'.
- #define delimQuot2 (0x27)

  Delimiter for Quoted Strings - Single Quote "'".
- #define delimQuot3 (0x60)

  Delimiter for Quoted Strings - Accent Grave '`'.
- #define cmdSerial ('S')

  Command Serial Number.
- #define cmdSequence ('s')

  Command Sequence Number.
- #define cmdVersion ('v')

  Command Version Number.
- #define cmdHardware ('H')

  Command Hardware Model String.
- #define cmdStyleCode ('h')

  Command Hardware Style Code in Hex.
- #define cmdID ('I')

  Command Identification String.
- #define cmdInterval ('i')

Command Interval Timer.

- #define cmdForward ('F')

  Command Forward text to PCD-bus.
- #define cmdReport ('r')

  Command Report Format.
- #define cmdDest ('R')

  Command Report Destination (like a phone number)
- #define cmdAnalog ('a')

  Command Analog Voltage.
- #define cmdMode ('m')

  Command Mode Bits.
- #define cmdMemory ('M')

  Command Memory Dump.
- #define cmdDate ('d')

  Command Date.
- #define cmdTime ('t')

  Command Time.
- #define cmdOperation ('o')

  Command Initiates the canned operations.
- #define cmdDiag ('z')

  Command Various Diagnostics.
- #define cmdPlaceholder ('%')

  Command (placeholder for future commands)
- #define SixtyTwo (0x24)

  Sixbit character 62 is ASCII '$'.
- #define SixtyThree (0x5F)

  Sixbit character 63 is ASCII '_'.
- enum uiState_t {
  none, up, down, enter,
  back, holding }

  ***uiState_t** Allowed States for Buttons or Scroll Wheel*
- enum cdState_t {
  cdWaitIdle, cdIdle, cdRecvData, cdTrmtData,
  cdRecvDone, cdTrmtDone }

  ***cdState_t** Allowed States for the PCD-Bus Interface*
- ModeBits_t ModeBits

  Non-Volatile shadow at nvMode (16 bits)
- SerialBits_t SerialBits

  Non-Volatile shadow at nvSerial (16 bits)
- ModbusBits_t ModbusBits

  Non-Volatile shadow at nvModbus (8 bits)
- FlagBits_t FlagBits

  Operating Status Flag Bits.
- CompareBits_t CompareBits

  Control and results of string comparisons.
- ErrorBits_t ErrorBits

  Global Error indication flags.

- #define EOF (0xFFFF)

  End of File signal for buffered reception.
- #define cdRingSize (150)

  Number of bytes allowed in cdRxRing and cdTxRing.
- #define rxRingSize (150)

  Number of bytes allowed in rxRing.
- #define txRingSize (150)

  Number of bytes allowed in txRing.
- #define msgRingSize (150)

  Number of bytes allowed in msgRing.
- #define scrRingSize (150)

  Number of bytes allowed in scrRing.
- #define resRingSize (150)

  Number of bytes allowed in resRing.
- uint8_t cdTxRing [cdRingSize]

  Ring Buffer for bytes to be transmitted over the PCD-Bus.
- uint8_t cdRxRing [cdRingSize]

  Ring Buffer for bytes being received over the PCD-Bus.
- uint8_t rxRing [rxRingSize]

  UART Receive Ring buffer.
- uint8_t txRing [txRingSize]

  Ring Buffer data automatically sent to the UART.
- uint8_t msgRing [msgRingSize]

  Message Ring buffer.
- uint8_t scrRing [scrRingSize]

  Scratch Ring buffer.
- uint8_t resRing [resRingSize]

  Response Ring buffer.
- ringSelect_t ringFrom

  Source for get() and peek()
- ringSelect_t ringTo

  Destination for put(ch)
- uint8_t cdRxGetPtr

  Index of next character to be read from the PCD-Bus Receive Ring.
- uint8_t cdRxPutPtr

  Index of next character to be placed in the PCD-Bus Receive Ring.
- uint8_t cdRxUsed

  Bytes used in cdRxRing.
- uint8_t cdTxGetPtr

  Index of next character to be sent from the PCD-Bus Transmit Ring.
- uint8_t cdTxPutPtr

  Index of next character to be placed in the PCD-Bus Transmit Ring.
- uint8_t cdTxUsed

  Bytes used in cdTxRing.
- uint8_t rxGetPtr

  index to get next byte from rxRing
- uint8_t rxPutPtr

Index to put next byte into rxRing.

- uint8_t rxUsed

  Bytes used in rxRing.

- uint8_t rxEcho

  Character to echo to the UART Transmit hardware.

- uint8_t txGetPtr

  Index of next character to send in the UART Transmit Ring.

- uint8_t txPutPtr

  Index of next character to append to the UART Transmit Ring.

- uint8_t txUsed

  Bytes used in txRing.

- uint8_t msgGetPtr

  index to get next byte from msgRing

- uint8_t msgPutPtr

  Index to put next byte into msgRing.

- uint8_t msgUsed

  Bytes used in msgRing.

- uint8_t scrGetPtr

  index to get next byte from scrRing

- uint8_t scrPutPtr

  Index to put next byte into scrRing.

- uint8_t scrUsed

  Bytes used in scrRing.

- uint8_t resGetPtr

  index to get next byte from resRing

- uint8_t resPutPtr

  Index to put next byte into resRing.

- uint8_t resUsed

  Bytes used in resRing.

12.2.1 Detailed Description

Control-Data Module Firmware.

Definition in file cdm.h.

12.2.2 Macro Definition Documentation

12.2.2.1 #define activeDelay {LATBbits.LB0 = 0; LATBbits.LB1 = 0; LATBbits.LB4 = 1; LATBbits.LB5 = 1; }

Definition at line 1337 of file cdm.h.

12.2.2.2 #define nvAvail1 (0x17)

8-bit unused =====

Definition at line 1009 of file cdm.h.

12.2.2.3 #define nvCmd0 (0x70)

Command to run when Interval 0 expires (32 bytes)

Definition at line 1021 of file cdm.h.

12.2.2.4 #define nvCmd1 (0x90)

Command to run when Interval 1 expires (32 bytes)

Definition at line 1022 of file cdm.h.

12.2.2.5 #define nvCmd2 (0xB0)

Command to run when Interval 2 expires (32 bytes)

Definition at line 1023 of file cdm.h.

12.2.2.6 #define nvDate (0x04)

32-Bit Manufacturing Date

Definition at line 1003 of file cdm.h.

12.2.2.7 #define nvDest0 (0x40)

Report Destination 0 (like a phone number)

Definition at line 1018 of file cdm.h.

12.2.2.8 #define nvDest1 (0x50)

Report Destination 1 (like a phone number)

Definition at line 1019 of file cdm.h.

12.2.2.9 #define nvDest2 (0x60)

Report Destination 2 (like a phone number)

Definition at line 1020 of file cdm.h.

12.2.2.10 #define nvID (0x20)

16-byte Device ID - User assignable

Definition at line 1016 of file cdm.h.

12.2.2.11 #define nvInt0 (0xD0)

Interval 0 Reload in Seconds (4 bytes)

Definition at line 1024 of file cdm.h.

12.2.2.12 #define nvInt1 (0xD4)

Interval 1 Reload in Seconds (4 bytes)

Definition at line 1025 of file cdm.h.

12.2.2.13 #define nvInt2 (0xD8)

Interval 2 Reload in Seconds (4 bytes)

Definition at line 1026 of file cdm.h.

12.2.2.14 #define nvModbus (0x1C)

8-bit ModBus Configuration Bits >> ModbusBits.w

Definition at line 1013 of file cdm.h.

12.2.2.15 #define nvMode (0x1A)

16-bit Operating Mode bits >> ModeBits.w

Definition at line 1012 of file cdm.h.

12.2.2.16 #define nvModel (0x30)

16-byte Device Model Number/Description

Definition at line 1017 of file cdm.h.

12.2.2.17 #define nvReset (0x18)

Count of Power-On Resets.

Definition at line 1010 of file cdm.h.

12.2.2.18 #define nvRTO (0x10)

32-bit Running Time Offset to UTC

Definition at line 1006 of file cdm.h.

12.2.2.19 #define nvSeq (0x19)

Sequence Number of last transmitted message.

Definition at line 1011 of file cdm.h.

12.2.2.20 #define nvSerial (0x1E)

16-bit Serial Port Configuration >> SerialBits.w

Definition at line 1015 of file cdm.h.

12.2.2.21 #define nvSlave (0x1D)

8-bit ModBus Slave Address (1-247 allowed)

Definition at line 1014 of file cdm.h.

12.2.2.22 #define nvSN (0x08)

32-bit Device Serial Number

Definition at line 1004 of file cdm.h.

12.2.2.23 #define nvStyle (0x16)

8-bit Board Style selects hardware feature drivers

Definition at line 1008 of file cdm.h.

12.2.2.24 #define nvTRT (0x0C)

32-bit Total Running Time (periodically updated)

Definition at line 1005 of file cdm.h.

12.2.2.25 #define nvZone (0x14)

16-bit Time Zone Offset in minutes

Definition at line 1007 of file cdm.h.

12.2.2.26 #define releaseC {TRISBbits.RB0 = 1; TRISBbits.RB4 = 1; if(ModeBits.fDiagLED)rightLED = 0; }

Definition at line 1340 of file cdm.h.

12.2.2.27 #define releaseD {TRISBbits.RB1 = 1; TRISBbits.RB5 = 1; if(ModeBits.fDiagLED)leftLED = 0; }

Definition at line 1341 of file cdm.h.

12.2.2.28 #define rtcReload (5859)

Reload Value for 8/sec no prescaler 48MHz / 1024.

Definition at line 1234 of file cdm.h.

12.2.2.29 #define setChigh {TRISBbits.RB0 = 1; TRISBbits.RB4 = 0; activeDelay; TRISBbits.RB4 = 1; if(ModeBits.fDiagLED)rightLED = 0; }

Definition at line 1342 of file cdm.h.

12.2.2.30 #define setClow {TRISBbits.RB0 = 0; activeDelay; if(ModeBits.fDiagLED)rightLED = 1; }

Definition at line 1344 of file cdm.h.

12.2.2.31 #define setDhigh {TRISBbits.RB1 = 1; TRISBbits.RB5 = 0; activeDelay; TRISBbits.RB5 = 1; if(ModeBits.fDiagLED)leftLED = 0; }

Definition at line 1343 of file cdm.h.

12.2.2.32 #define setDlow {TRISBbits.RB1 = 0; activeDelay; if(ModeBits.fDiagLED)leftLED = 1; }

Definition at line 1345 of file cdm.h.

12.2.2.33 #define testC (PORTBbits.RB0)

Definition at line 1338 of file cdm.h.

12.2.2.34 #define testD (PORTBbits.RB1)

Definition at line 1339 of file cdm.h.

12.2.2.35 #define uiButtonMask (uiUp | uiDown | uiEnter)

Mask for input bits in port.

Definition at line 1138 of file cdm.h.

12.2.2.36 #define uiButtons (PORTC)

PIC Port containing User Interface input bits.

Definition at line 1137 of file cdm.h.

12.2.2.37 #define uiDown (0x04)

Input bit mask for the DOWN button.

Definition at line 1141 of file cdm.h.

12.2.2.38 #define uiEnter (0x02)

Input bit mask for the ENTER button.

Definition at line 1140 of file cdm.h.

12.2.2.39 #define uiUp (0x01)

Input bit mask for the UP button.

Definition at line 1139 of file cdm.h.

12.2.3 Function Documentation

12.2.3.1 uint16_t usbGet (void)

usbGet ...

Returns

12.2.3.2 uint16_t usbPeek (int16_t *n*)

usbPeek ...

Parameters

n	

Returns

12.2.3.3 uint16_t usbPut (uint16_t *ch*)

usbPut ...

Parameters

ch	

Returns

12.2.4 Variable Documentation

12.2.4.1 uint8_t adcDisplayInterval

Display interval for ADC results, if used.

Definition at line 1184 of file cdm.h.

12.2.4.2 uint16_t adcResult[3]

Current values of the results from each ADC channel.

ingroup ADC

Definition at line 1183 of file cdm.h.

12.2.4.3 uint8_t cdDiagRB

Received message count.

Definition at line 1330 of file cdm.h.

12.2.4.4 uint8_t cdDiagRD

Received message count (discarded)

Definition at line 1329 of file cdm.h.

12.2.4.5 uint8_t cdDiagRE

Received message Error count.

Definition at line 1331 of file cdm.h.

12.2.4.6 uint8_t cdDiagTB

Transmitted message count.

Definition at line 1332 of file cdm.h.

12.2.4.7 uint8_t cdDiagTE

Transmitted message Error count.

Definition at line 1333 of file cdm.h.

12.2.4.8 cdState_t cdState = cdWaitIdle

Definition at line 1335 of file cdm.h.

12.2.4.9 uint8_t diCursor = 0xFF

Cursor Position (0..31), or 0xFF for reset.

Definition at line 1134 of file cdm.h.

12.2.4.10 uint8_t diDisplay[32]

LCD Display Buffer. Size must be a power of two.

Definition at line 1136 of file cdm.h.

12.2.4.11 uint32_t lastCommandSender

Definition at line 1412 of file cdm.h.

12.2.4.12 uint8_t ledA

Current flashing pattern for Left LED.

Definition at line 1229 of file cdm.h.

12.2.4.13 uint8_t ledAreload

Default flashing pattern for Left LED.

Definition at line 1231 of file cdm.h.

12.2.4.14 uint8_t ledB

Current flashing pattern for Rigt LED.

Definition at line 1230 of file cdm.h.

12.2.4.15 uint8_t ledBreload

Default flashing pattern for Right LED.

Definition at line 1232 of file cdm.h.

12.2.4.16 uint8_t ledMask

Bit mask for LED blinking pattern.

Definition at line 1228 of file cdm.h.

12.2.4.17 uint16_t pollLongest

Longest Critical Polling interval.

Definition at line 1249 of file cdm.h.

12.2.4.18 uint32_t rtcClock

Local Clock Time in Seconds within current day.

Definition at line 1221 of file cdm.h.

12.2.4.19 uint8_t rtcDay

Local Time Calendar Day (1-31)

Definition at line 1219 of file cdm.h.

12.2.4.20 uint8_t rtcDOW

Local Time Day of week (1-7, Sun-Sat)

Definition at line 1220 of file cdm.h.

12.2.4.21 uint8_t rtcMonth

Local Time Calendar Month (1-12)

Definition at line 1218 of file cdm.h.

12.2.4.22 uint8_t rtcTicks

Counts Fast interrupts for sub-second timing (0..7) >= 8 when a second passes.

Definition at line 1227 of file cdm.h.

12.2.4.23 uint32_t rtcTimerA

One of three general-purpose down-counters (in seconds)

Definition at line 1223 of file cdm.h.

12.2.4.24 uint32_t rtcTimerB

One of three general-purpose down-counters (in seconds)

Definition at line 1224 of file cdm.h.

12.2.4.25 uint32_t rtcTimerC

One of three general-purpose down-counters (in seconds)

Definition at line 1225 of file cdm.h.

12.2.4.26 uint8_t rtcYear

Local Time Calendar Year (14-99)

Definition at line 1217 of file cdm.h.

12.2.4.27 uint32_t rxMicroSeconds

Precision time of last received character.

Definition at line 201 of file cdm.h.

12.2.4.28 uint8_t uiChanged

Bits that just changed.

Definition at line 1167 of file cdm.h.

12.2.4.29 uint8_t uiCurrent

Most recently read button bit pattern.

Definition at line 1166 of file cdm.h.

12.2.4.30 uint8_t uiCursor = 0xFF

Position of data entry cursor.

Definition at line 1135 of file cdm.h.

12.2.4.31 uint8_t uiHeld

Ticks since last change.

Definition at line 1169 of file cdm.h.

12.2.4.32 uint8_t uiLast

Previous button bit pattern.

Definition at line 1168 of file cdm.h.

12.2.4.33 int8_t uiScroll

Signed running total of Up and Down counts.

Definition at line 1170 of file cdm.h.

12.2.4.34 uiState_t uiState = none

Definition at line 1164 of file cdm.h.

12.2.4.35 uint8_t uiTicks

Tick counter to time button-held-down.

Definition at line 1165 of file cdm.h.

12.3 source/LaserSensorUserGuide.md File Reference

12.4 source/main.c File Reference

```
#include <xc.h>
#include <string.h>
#include <plib\timers.h>
#include "cdm.h"
#include "main.h"
```

Functions

- int16_t main (void)

  main
- void interrupt high_priority isr ()

  high_priority *High Priority Interrupt*

Variables

- volatile unsigned char WPUB

12.4.1 Function Documentation

12.4.1.1 int16_t main (void)

main

The Universal Intelligent Sensor Controller provides a standardized method of sensing, reporting and controlling a variety of devices.

Returns

Definition at line 92 of file main.c.

```
93 {   // **** ... **main** The Real, True Main Program
94     uint16_t ch;
95
96     OSCCONbits.IRCF = 7; // This should be 16MHz internal
97     OSCCON2bits.PLLEN = 1; // Apparently we need the PLL in 3X mode for 48MHz
98
99
100     ANSELA = 0x07; // Disable the Analog Inputs of PORTA Leave 3 Analog Inputs
101     ANSELB = 0; // Disable the Analog Inputs of PORTB
102     ANSELC = 0; // Disable the Analog Inputs of PORTC
103     LATA = 0;
104     TRISA = 0x07; // PORTA outputs for LEDs and PSon
105     TRISB = 0xFF; // PORTB is all inputs while idle. No effect on CD-Bus
106     INTCON2bits.RBPU = 0; // Allow Weak Pull-ups on PortB
107     WPUB = 0x03; // Weak Pullups only on the C and D lines
108     LATC = 0x80;
109     TRISC = 0x80; // PORTC is driven low except for the USART RX line
110
111     CTSintris = 1;
112     DSRintris = 1;
113     RTSouttris = 0;
114
115     // Internal Clock is 12MHz (48MHz / 4)
116     T0CON = 0x87; // Timer 0 counts 16-bits of internal clock /1024 using prescaler
117     INTCONbits.TMR0IF = 0;  // Reset the timer and prescaler and clear the interrupt
118     INTCONbits.TMR0IE = 1;  // Enable Timer Interrupts
119
120     // Set up the USART and send a character
121     PIE1bits.TX1IE = 0; // No transmit interrupts
122     PIE1bits.RCIE = 0; // No receive interrupts
123
124     eeInit();
125     ch = eeGet(nvReset);
126     eePut(ch+1);   // Increment the non-volatile Reset Count
127
128     ensureBaud(SerialBits.baud);    // This is the baud rate selector
129     rtcGetSavedParams();
130
131     INTCONbits.PEIE = 1;    // Enable Peripheral Device interrupts
132     INTCONbits.GIE = 1;     // Turn on the interrupts and we are running
133
134     initMessage();
135
136     while(1) {
137         wait(0);    // Must do polling
138         if (eeGet32(nvSN) == 0xFFFFFFFF) Virgin(); else  // Must have S/N to be non-virgin
139         switch ((style_t) eeGet(nvStyle)) {    // Get the operating board style
140             case styleUser:     User();         break;
141             case styleCDMA:     CDMA();         break;
142             case styleGSM:      GSM();          break;
143             case styleLaser:    Laser();        break;
144             case styleUSB:      USB();          break;
145             case styleRS485:    RS485();        break;
146             case styleModbus:   Modbus();       break;
147             case styleSerial:   Serial();       break;
148             case styleTest:     Test();         break;
149             case styleBlue:
150             case styleWiFi:
151             case styleSNAP:
152             case styleVirgin:
```

```
            default:            Virgin();
        }
    }

    return 0;
}
```

12.4.2 Variable Documentation

12.4.2.1 volatile unsigned char WPUB

The Universal Intelligent Sensor Controller provides a standardized method of sensing, reporting and controlling a variety of devices.

12.5 source/main.h File Reference

Control-Data Module Main Program.

Functions

- int16_t main (void)

  main
- void interrupt high_priority isr ()

  high_priority *High Priority Interrupt*

12.5.1 Detailed Description

Control-Data Module Main Program.

Definition in file main.h.

12.5.2 Function Documentation

12.5.2.1 int16_t main (void)

main

The Universal Intelligent Sensor Controller provides a standardized method of sensing, reporting and controlling a variety of devices.

Returns

Definition at line 92 of file main.c.

```
93 {   // **** ... **main** The Real, True Main Program
94     uint16_t ch;
95
96     OSCCONbits.IRCF = 7; // This should be 16MHz internal
97     OSCCON2bits.PLLEN = 1; // Apparently we need the PLL in 3X mode for 48MHz
98
99
```

```
    ANSELA = 0x07; // Disable the Analog Inputs of PORTA Leave 3 Analog Inputs
    ANSELB = 0; // Disable the Analog Inputs of PORTB
    ANSELC = 0; // Disable the Analog Inputs of PORTC
    LATA = 0;
    TRISA = 0x07; // PORTA outputs for LEDs and PSon
    TRISB = 0xFF; // PORTB is all inputs while idle. No effect on CD-Bus
    INTCON2bits.RBPU = 0; // Allow Weak Pull-ups on PortB
    WPUB = 0x03; // Weak Pullups only on the C and D lines
    LATC = 0x80;
    TRISC = 0x80; // PORTC is driven low except for the USART RX line

    CTSintris = 1;
    DSRintris = 1;
    RTSouttris = 0;

    // Internal Clock is 12MHz (48MHz / 4)
    T0CON = 0x87; // Timer 0 counts 16-bits of internal clock /1024 using prescaler
    INTCONbits.TMR0IF = 0;  // Reset the timer and prescaler and clear the interrupt
    INTCONbits.TMR0IE = 1;  // Enable Timer Interrupts

    // Set up the USART and send a character
    PIE1bits.TX1IE = 0; // No transmit interrupts
    PIE1bits.RCIE = 0; // No receive interrupts

    eeInit();
    ch = eeGet(nvReset);
    eePut(ch+1);   // Increment the non-volatile Reset Count

    ensureBaud(SerialBits.baud);    // This is the baud rate selector
    rtcGetSavedParams();

    INTCONbits.PEIE = 1;    // Enable Peripheral Device interrupts
    INTCONbits.GIE = 1;     // Turn on the interrupts and we are running

    initMessage();

    while(1) {
        wait(0);    // Must do polling
        if (eeGet32(nvSN) == 0xFFFFFFFF) Virgin(); else  // Must have S/N to be non-virgin
        switch ((style_t) eeGet(nvStyle)) {    // Get the operating board style
            case styleUser:     User();         break;
            case styleCDMA:     CDMA();         break;
            case styleGSM:      GSM();          break;
            case styleLaser:    Laser();        break;
            case styleUSB:      USB();          break;
            case styleRS485:    RS485();        break;
            case styleModbus:   Modbus();       break;
            case styleSerial:   Serial();       break;
            case styleTest:     Test();         break;
            case styleBlue:
            case styleWiFi:
            case styleSNAP:
            case styleVirgin:
            default:            Virgin();
        }
    }

    return 0;
}
```

12.6 source/MainPage.md File Reference

12.7 source/Manual.md File Reference

12.8 source/ModBus.md File Reference

12.9 source/TxAM-AdvancedPumpController.md File Reference

12.10 source/UllageSensorUserGuide.md File Reference

12.11 source/usb.c File Reference

```
#include <string.h>
#include "usb_config.h"
#include "usb.h"
#include "usb_hal.h"
#include "usb_ch9.h"
#include "usb_microsoft.h"
#include "usb_winusb.h"
```

Data Structures

- struct ep_buf
- struct ep0_buf

Macros

- #define MIN(x, y) (((x)<(y))?(x):(y))
- #define EP_0_OUT_LEN EP_0_LEN
- #define EP_0_IN_LEN EP_0_LEN
- #define PPB_EP0_OUT
- #define NUM_BD_0 3
- #define NUM_BD (2 ∗ (NUM_ENDPOINT_NUMBERS) + NUM_BD_0)
- #define BDS0OUT(oe) bds[0 + oe]
- #define BDS0IN(oe) bds[2]
- #define BDSnOUT(EP, oe) bds[(EP) ∗ 2 + 1]
- #define BDSnIN(EP, oe) bds[(EP) ∗ 2 + 2]
- #define EP_BUF(n)
- #define EP_BUF(n)
- #define EP_OUT_HALT_FLAG 0x1
- #define EP_IN_HALT_FLAG 0x2
- #define EP_RX_DTS 0x4 /∗ The DTS of the _next_ packet ∗/
- #define EP_TX_DTS 0x8
- #define EP_RX_PPBI
- #define EP_TX_PPBI 0x20 /∗ Represents the _next_ buffer to write into. ∗/
- #define EP_BUFS0()
- #define EP_BUFS(n)
- #define SERIAL(x)
- #define SERIAL_VAL(x)
- #define copy_to_ep0_in_buf(PTR, LEN) memcpy_from_rom(ep0_buf.in, PTR, LEN);

Functions

- STATIC_SIZE_CHECK_EQUAL (sizeof(struct endpoint_descriptor), 7)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct interface_descriptor), 9)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct configuration_descriptor), 9)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct device_descriptor), 18)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct interface_association_descriptor), 8)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct setup_packet), 8)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_os_descriptor), 18)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_compat_header), 16)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_compat_function), 24)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_properties_header), 10)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_property_section_header), 8)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct buffer_descriptor), 4)
- void usb_init (void)

  Initialize the USB library and hardware.
- void usb_service (void)

  Update the USB library and hardware.
- uint8_t usb_get_configuration (void)

  Get the device configuration.
- unsigned char * usb_get_in_buffer (uint8_t endpoint)

  Get a pointer to an endpoint's input buffer.
- void usb_send_in_buffer (uint8_t endpoint, size_t len)

  Send an endpoint's IN buffer to the host.
- bool usb_in_endpoint_busy (uint8_t endpoint)

  Check whether an IN endpoint is busy.
- bool usb_in_endpoint_halted (uint8_t endpoint)

  Check whether an endpoint is halted.
- uint8_t usb_get_out_buffer (uint8_t endpoint, const unsigned char **buf)

  Get a pointer to an endpoint's OUT buffer.
- bool usb_out_endpoint_has_data (uint8_t endpoint)

  Check whether an OUT endpoint has received data.
- void usb_arm_out_endpoint (uint8_t endpoint)

  Re-enable reception on an OUT endpoint.
- bool usb_out_endpoint_halted (uint8_t endpoint)

  Check whether an OUT endpoint is halted.
- void usb_start_receive_ep0_data_stage (char *buffer, size_t len, usb_ep0_data_stage_callback callback, void *context)

  Start the data stage of an OUT control transfer.
- void usb_send_data_stage (char *buffer, size_t len, usb_ep0_data_stage_callback callback, void *context)

  Start the data stage of an IN control transfer.

12.11.1 Macro Definition Documentation

12.11.1.1 #define BDS0IN(*oe*) bds[2]

Definition at line 152 of file usb.c.

12.11.1.2 #define BDS0OUT(*oe*) bds[0 + oe]

Definition at line 151 of file usb.c.

12.11.1.3 #define BDSnIN(*EP, oe*) bds[(EP) ∗ 2 + 2]

Definition at line 154 of file usb.c.

12.11.1.4 #define BDSnOUT(*EP, oe*) bds[(EP) ∗ 2 + 1]

Definition at line 153 of file usb.c.

12.11.1.5 #define copy_to_ep0_in_buf(*PTR, LEN*) memcpy_from_rom(ep0_buf.in, PTR, LEN);

Definition at line 776 of file usb.c.

12.11.1.6 #define EP_0_IN_LEN EP_0_LEN

Definition at line 68 of file usb.c.

12.11.1.7 #define EP_0_OUT_LEN EP_0_LEN

Definition at line 67 of file usb.c.

12.11.1.8 #define EP_BUF(*n*)

Value:

```
unsigned char ep_##n##_out_buf[2][EP_##n##_OUT_LEN]; \
        unsigned char ep_##n##_in_buf[1][EP_##n##_IN_LEN];
```

Definition at line 239 of file usb.c.

12.11.1.9 #define EP_BUF(*n*)

Value:

```
unsigned char ep_##n##_out_buf[1][EP_##n##_OUT_LEN]; \
        unsigned char ep_##n##_in_buf[1][EP_##n##_IN_LEN];
```

Definition at line 239 of file usb.c.

12.11.1.10 #define EP_BUFS(*n*)

Value:

```
{ ep_buffers.ep_##n##_out_buf[0], \
                        ep_buffers.ep_##n##_in_buf[0], \
                        EP_##n##_OUT_LEN, \
                        EP_##n##_IN_LEN },
```

Definition at line 359 of file usb.c.

12.11.1.11 #define EP_BUFS0( )

Value:

```
{ ep_buffers.ep_0_out_buf[0], \
                        ep_buffers.ep_0_in_buf[0], \
                        ep_buffers.ep_0_out_buf[1] }
```

Definition at line 338 of file usb.c.

12.11.1.12 #define EP_IN_HALT_FLAG 0x2

Definition at line 304 of file usb.c.

12.11.1.13 #define EP_OUT_HALT_FLAG 0x1

Definition at line 303 of file usb.c.

12.11.1.14 #define EP_RX_DTS 0x4 /∗ The DTS of the _next_ packet ∗/

Definition at line 305 of file usb.c.

12.11.1.15 #define EP_RX_PPBI

Value:

```
0x10 /* Represents the next buffer which will be need to be
                          reset and given back to the SIE. */
```

Definition at line 307 of file usb.c.

12.11.1.16 #define EP_TX_DTS 0x8

Definition at line 306 of file usb.c.

12.11.1.17 #define EP_TX_PPBI 0x20 /∗ Represents the _next_ buffer to write into. ∗/

Definition at line 309 of file usb.c.

12.11.1.18 #define MIN(*x*, *y*) (((x)<(y))?(x):(y))

Definition at line 62 of file usb.c.

12.11.1.19 #define NUM_BD (2 ∗ (NUM_ENDPOINT_NUMBERS) + NUM_BD_0)

Definition at line 142 of file usb.c.

12.11.1.20 #define NUM_BD_0 3

Definition at line 132 of file usb.c.

12.11.1.21 #define PPB_EP0_OUT

Definition at line 82 of file usb.c.

12.11.1.22 #define SERIAL(*x*)

Definition at line 481 of file usb.c.

12.11.1.23 #define SERIAL_VAL(*x*)

Definition at line 482 of file usb.c.

12.11.2 Function Documentation

12.11.2.1 STATIC_SIZE_CHECK_EQUAL (sizeof(struct endpoint_descriptor) , 7)

12.11.2.2 STATIC_SIZE_CHECK_EQUAL (sizeof(struct interface_descriptor) , 9)

12.11.2.3 STATIC_SIZE_CHECK_EQUAL (sizeof(struct configuration_descriptor) , 9)

12.11.2.4 STATIC_SIZE_CHECK_EQUAL (sizeof(struct device_descriptor) , 18)

12.11.2.5 STATIC_SIZE_CHECK_EQUAL (sizeof(struct interface_association_descriptor) , 8)

12.11.2.6 STATIC_SIZE_CHECK_EQUAL (sizeof(struct setup_packet) , 8)

12.11.2.7 STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_os_descriptor) , 18)

12.11.2.8 STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_compat_header) , 16)

12.11.2.9 STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_compat_function) , 24)

12.11.2.10 STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_properties_header) , 10)

12.11.2.11 STATIC_SIZE_CHECK_EQUAL (sizeof(struct microsoft_extended_property_section_header) , 8)

12.11.2.12 STATIC_SIZE_CHECK_EQUAL (sizeof(struct buffer_descriptor) , 4)

12.12 source/usb.h File Reference

USB Protocol Stack.

```
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include "usb_config.h"
```

Macros

- #define usb_is_configured() (usb_get_configuration() != 0)

  Determine whether the device is in the Configured state.

Typedefs

- typedef void(∗ usb_ep0_data_stage_callback)(bool transfer_ok, void ∗context)

  Endpoint 0 data stage callback definition.

Functions

- int16_t USB_STRING_DESCRIPTOR_FUNC (uint8_t string_number, const void ∗∗ptr)
- void usb_init (void)

  Initialize the USB library and hardware.
- void usb_service (void)

  Update the USB library and hardware.
- uint8_t usb_get_configuration (void)

  Get the device configuration.
- unsigned char ∗ usb_get_in_buffer (uint8_t endpoint)

  Get a pointer to an endpoint's input buffer.
- void usb_send_in_buffer (uint8_t endpoint, size_t len)

  Send an endpoint's IN buffer to the host.
- bool usb_in_endpoint_busy (uint8_t endpoint)

  Check whether an IN endpoint is busy.
- bool usb_in_endpoint_halted (uint8_t endpoint)

  Check whether an endpoint is halted.
- bool usb_out_endpoint_has_data (uint8_t endpoint)

  Check whether an OUT endpoint has received data.
- void usb_arm_out_endpoint (uint8_t endpoint)

  Re-enable reception on an OUT endpoint.
- bool usb_out_endpoint_halted (uint8_t endpoint)

  Check whether an OUT endpoint is halted.
- uint8_t usb_get_out_buffer (uint8_t endpoint, const unsigned char ∗∗buffer)

  Get a pointer to an endpoint's OUT buffer.
- void usb_start_receive_ep0_data_stage (char ∗buffer, size_t len, usb_ep0_data_stage_callback callback, void ∗context)

  Start the data stage of an OUT control transfer.
- void usb_send_data_stage (char ∗buffer, size_t len, usb_ep0_data_stage_callback callback, void ∗context)

  Start the data stage of an IN control transfer.

Variables

- const struct device_descriptor USB_DEVICE_DESCRIPTOR
- const struct configuration_descriptor ∗ USB_CONFIG_DESCRIPTOR_MAP []

12.12.1 Detailed Description

USB Protocol Stack.

Definition in file usb.h.

12.13 source/usb_cdc.c File Reference

```
#include "usb_config.h"
#include "usb_ch9.h"
#include "usb.h"
#include "usb_cdc.h"
```

Data Structures

- union **transfer_data**

Macros

- #define MIN(x, y) (((x)<(y))?(x):(y))

Functions

- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_functional_descriptor_header), 5)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_acm_functional_descriptor), 4)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_union_functional_descriptor), 5)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_line_coding), 7)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_notification_header), 8)
- STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_serial_state_notification), 10)
- uint8_t process_cdc_setup_request (const struct setup_packet *setup)

12.13.1 Macro Definition Documentation

12.13.1.1 #define MIN(*x*, *y*) (((x)<(y))?(x):(y))

Definition at line 33 of file usb_cdc.c.

12.13.2 Function Documentation

12.13.2.1 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_functional_descriptor_header) , 5)

12.13.2.2 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_acm_functional_descriptor) , 4)

12.13.2.3 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_union_functional_descriptor) , 5)

12.13.2.4 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_line_coding) , 7)

12.13.2.5 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_notification_header) , 8)

12.13.2.6 STATIC_SIZE_CHECK_EQUAL (sizeof(struct cdc_serial_state_notification) , 10)

12.14 source/usb_cdc.h File Reference

USB CDC Class Enumerations and Structures.

```
#include <stdint.h>
#include "usb_config.h"
```

Data Structures

- struct cdc_functional_descriptor_header
- struct cdc_acm_functional_descriptor
- struct cdc_union_functional_descriptor
- struct cdc_notification_header
- struct cdc_serial_state_notification
- struct cdc_line_coding

Macros

- #define CDC_DEVICE_CLASS 0x02 /∗ 4.1 ∗/
- #define CDC_COMMUNICATION_INTERFACE_CLASS 0x02 /∗ 4.2 ∗/
- #define CDC_COMMUNICATION_INTERFACE_CLASS_ACM_SUBCLASS 0x02 /∗ 4.3 ∗/
- #define CDC_DATA_INTERFACE_CLASS 0x0a /∗ 4.5 ∗/
- #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_NONE 0x0 /∗ 4.7 ∗/
- #define CDC_DATA_INTERFACE_CLASS_PROTOCOL_VENDOR 0xff /∗ 4.7 ∗/

Enumerations

- enum CDCDescriptorTypes { DESC_CS_INTERFACE = 0x24, DESC_CS_ENDPOINT = 0x25 }
- enum CDCFunctionalDescriptorSubtypes { CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_HEADER = 0x0, CDC_FUNCTIONAL_DESCRIPTOR_SUBTYPE_ACM = 0x2, CDC_FUNCTIONAL_DESCRIPTOR_SUBTYP↩E_UNION = 0x6 }
- enum CDCACMCapabilities { CDC_ACM_CAPABILITY_COMM_FEATURES = 0x1, CDC_ACM_CAPABILITY↩_LINE_CODINGS = 0x2, CDC_ACM_CAPABILITY_SEND_BREAK = 0x4, CDC_ACM_CAPABILITY_NETWO↩RK_CONNECTION = 0x8 }
- enum CDCRequests { CDC_SEND_ENCAPSULATED_COMMAND = 0x0, CDC_GET_ENCAPSULATED_RESPONSE = 0x1, CDC_↩SET_COMM_FEATURE = 0x2, CDC_GET_COMM_FEATURE = 0x3, CDC_CLEAR_COMM_FEATURE = 0x4, CDC_SET_LINE_CODING = 0x20, CDC_GET_LINE_CODING = 0x21, CDC_SET_CONTROL_LINE_STATE = 0x22, CDC_SEND_BREAK = 0x23 }
- enum CDCCommFeatureSelector { CDC_FEATURE_ABSTRACT_STATE = 0x1, CDC_FEATURE_COUNTR↩Y_SETTING = 0x2 }
- enum CDCCharFormat { CDC_CHAR_FORMAT_1_STOP_BIT = 0, CDC_CHAR_FORMAT_1_POINT_5_ST↩OP_BITS = 1, CDC_CHAR_FORMAT_2_STOP_BITS = 2 }

- enum CDCParityType {
  CDC_PARITY_NONE = 0, CDC_PARITY_ODD = 1, CDC_PARITY_EVEN = 2, CDC_PARITY_MARK = 3, CDC_PARITY_SPACE = 4 }
- enum CDCNotifications { CDC_NETWORK_CONNECTION = 0x0, CDC_RESPONSE_AVAILABLE = 0x1, CDC_SERIAL_STATE = 0x20 }

Functions

- uint8_t process_cdc_setup_request (const struct setup_packet *setup)
- int8_t CDC_SEND_ENCAPSULATED_COMMAND_CALLBACK (uint8_t interface, uint16_t length)

12.14.1 Detailed Description

USB CDC Class Enumerations and Structures.

Definition in file usb_cdc.h.