MPR GUI Release 6

by: Daniel M. Dobkin, Enigmatics

MPR GUI is a user interface for WJ Communications' MPR PC-card RFID readers. This software and documentation is released open-source; you may use and modify the software and documentation as you find appropriate, so long as the attribution statement remains in the code, and the Enigmatics logo remains in the documents.

Enigmatics makes no warranty regarding the suitability of this software for any specific application.

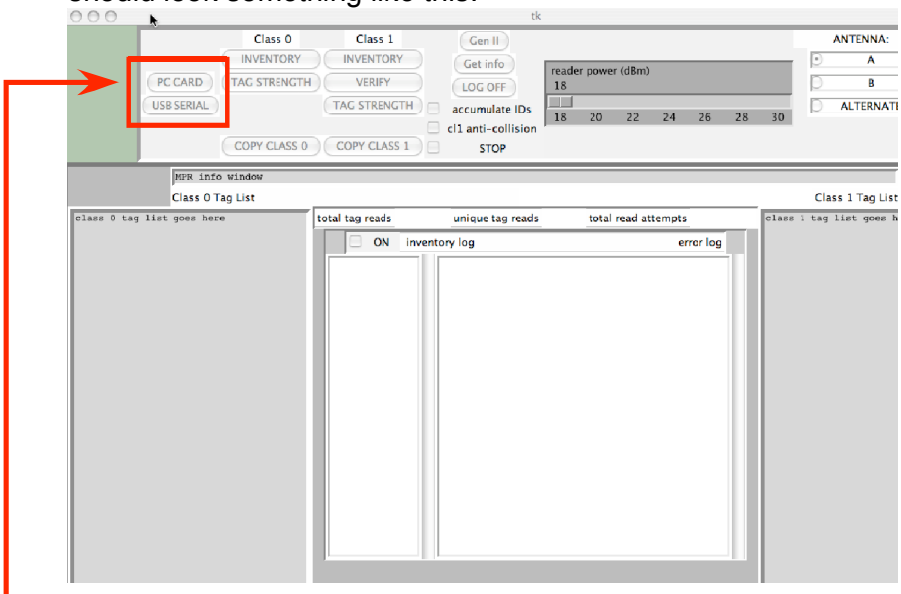
I would like to thank the people at WJ Communications for their support and assistance in writing and testing the software.

Part 1: User's Guide

To run the software you need to have Python resident on your machine. Mac OS X 10.2 or above comes with Python 2.3 resident, though the MacPython development environment is a nice addition. Linux or Windows machines may need to add Python. I'm running version 2.3 and haven't tested 2.4. You also need pyserial (to talk to the serial port device) and Tkinter to generate the user interface. Both of these are widely available Python distributions and may already be resident on your machine.

To run the code, make sure all the modules (files ending in .py) are in one folder, or in folders listed in your Python path variable. You can't run MPRGUI from inside an integrated development environment (IDE), as the IDE and the GUI will argue about ownership of the graphical interface. You can run the program from the terminal command line, or get something like Python Launcher for drag-and-drop:

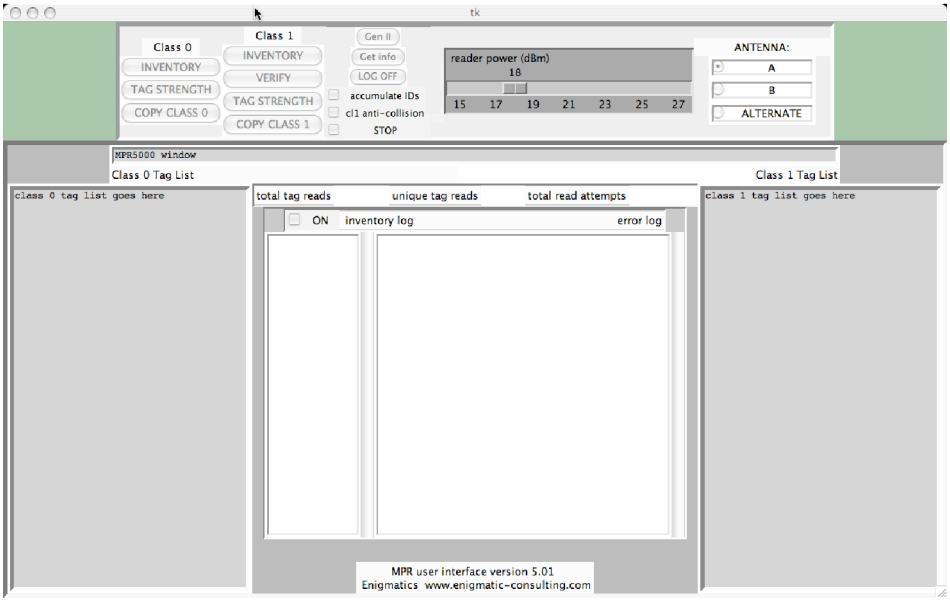
Drop the file MPRGUI_6.0.py on the Launcher or launch from the terminal. You will first see a terminal window and then the graphical interface will come up. It should look something like this:



The first thing you need to do is to press either the PC card or USB serial button, depending on how your MPR card is installed. [Right now these buttons are configured for a Mac PC-card slot and a Keyspan serial USB adaptor. To change the configuration you need to look in the dev library, usually named /dev,

and see what file is added when you insert the card or connect the serial adaptor. If you are working on a Windows machine, look in the System:Hardware control panel tab under the serial port heading and see what COM port your hardware has enumerated to. You then need to go to the routine Set_serial_port in reader_functions.py and change the names of the relevant devices.]

After you press one button or the other, the interface will attempt to get an info packet from the MPR. If this is the first time you've tried since you powered the MPR up, it often sends a number of bad packets first; be patient. If there are errors they will appear in the terminal window. After a few seconds, the serial port buttons will disappear and the interface window will look like this:



Press the INVENTORY button for either EPCglobal class 0 or class 1 tags to read them; the MPR will then continue to read tags until you press the STOP button. You can adjust the transmit power using the slider, and the antenna selection using the radio buttons; these settings are only registered at the start of an inventory, and ignored while the reader is reading. If you chose ALTERNATE the reader will switch between antenna A and antenna B in each inventory. (An MPR5000 has only antenna A and B or alternate don't do anything.)

If 'accumulate IDs' is cleared, the list of tags read after every 10 inventories will be displayed, and rewritten for the next burst of 10 inventories. If 'accumulate IDs' is checked, the list will accumulate each new tag for as long as the inventory sequence continues. If the list grows longer than the text field can display you can click within the text and drag the cursor down to see additional tag IDs.

If inventory log ON is checked, the interface will report the number of tags read in each inventory in a scrollable window. If the MPR reader sends error messages (e.g. 'antenna fault' if you forgot to connect the antenna), these will appear along with the sequential number of the bad inventory in the error log window.

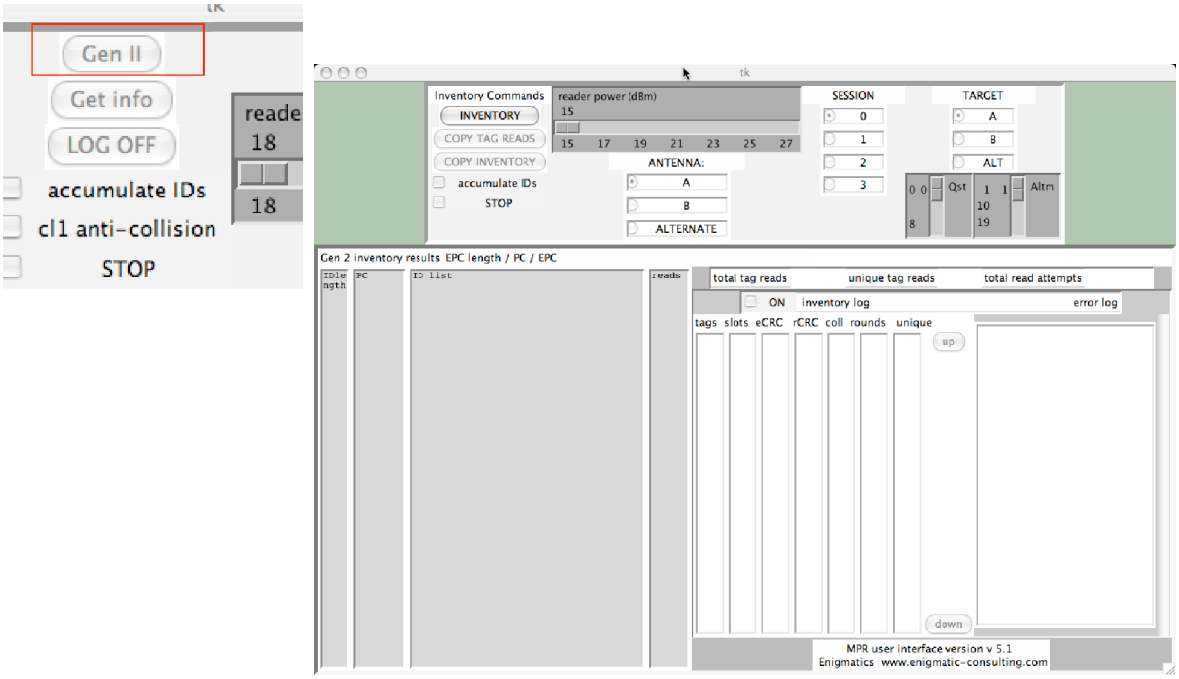
Class 0 and class 1 handle multiple-tag reading differently. The MPRGUI defaults to using ID2 (the tag ID) to singulate class 0 tags, and SCROLLALL to read class 1 tags. SCROLLALL will work poorly if more than 3 or 4 tags are in the field; in this case, you should check 'cl1 anti-collision' before starting an inventory. This will run a proprietary version of the PING algorithm, which is slow but can read up to > 30 tags in a single inventory.

TAG STRENGTH makes a series of inventories starting at the highest transmit power allowed in your system and decreasing until the minimum is reached or no tags are read. The results are reported in the relevant text field. This iis a very quick way to evaluate whether a tag is reading well or marginally, useful for tags on nasty surfaces like boxes with metallic or aqueous objects inside.

COPY CLASS 0 and COPY CLASS 1 buttons place the contents of the cited tag ID field onto the clipboard, for quick pasting into a word processor, spreadsheet, or other program. All the text fields can be selected and copied manually using your mouse.

Pressing the LOG OFF button will turn the Log on; every packet exchange is then recorded in the default output device, typically the terminal window.

Pressing the Gen 2 button brings up the Gen 2 interface window.

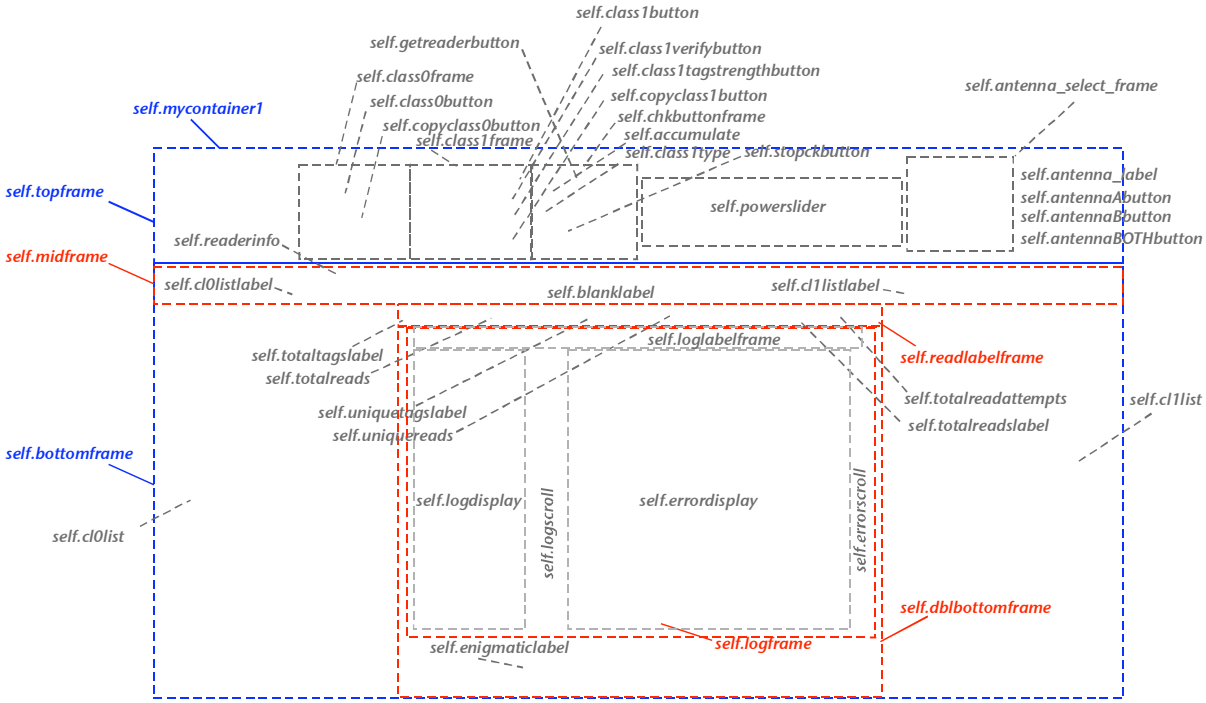


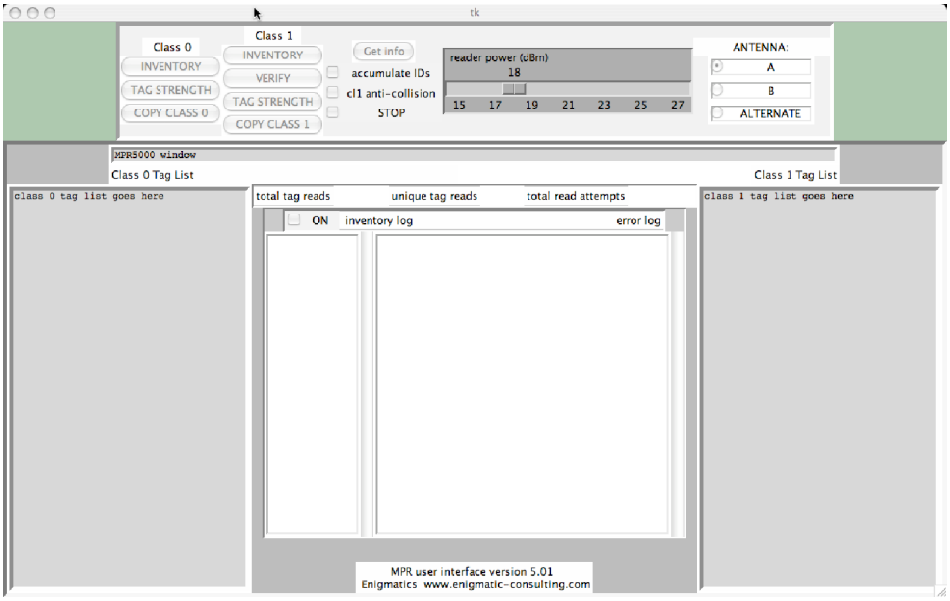
The interface is similar to the Gen 1 interface but with some new features. Session and Target are described in the Gen 2 standard; when the target is set to Alt, Alm inventories are taken with e.g. target A, and then Altm inventories are taken with target B, where Altm is set by the slider below the target selection buttons. Qstart is the initial value of the parameter Q that determines inventory size.

Inventory reports the length of the EPC, the PC (protocol control) word, and the EPC, as well as the number of times each EPC is read (if accumulate is checked). On the right side we get the summary values of unique tag reads and total reads and attempts. If the inventory log is checked, you also get a run-by-run log: tags read, slots used, EPC CRC and response CRC errors, collisions, rounds, and the running log of unique tags read. Finally, an error log reports packet errors (e.g. antenna faults) when they occur.

MPR GUI is written in Python 2.3, and works on my Mac under OS 10.3. Since this was my first (!) Python project the structure is VERY simple and not very elegant; only one class definition, for the user interface, is present. The code is divided amongst several modules, generally serving the functions of calculating CRCs, assembling commands, massaging packets, talking with the reader, and providing the user interface.

The interface uses Tkinter, which is reputed to be reasonably cross-platform. In Tkinter, the root window contains frames, and the frames contain widgets such as buttons, sliders, and text fields. Thus the MPRGUI class is a long list of these objects. The diagram below depicts roughly how the various widgets map into the user interface you see.





As described in the MPR API, the reader never sends a packet except in response to a command. The responsed packets are either intermediate packets or final packets, as shown at right.

Intermediate packets

SOF	0x01
node	0x00
packet length	varies
status	0x01
number of IDs	1 byte
tag ID 1	8 or 12 bytes
...	
tag ID n	8-12 bytes
CRC MSB	CRC MSB
CRC LSB	CRC LSB

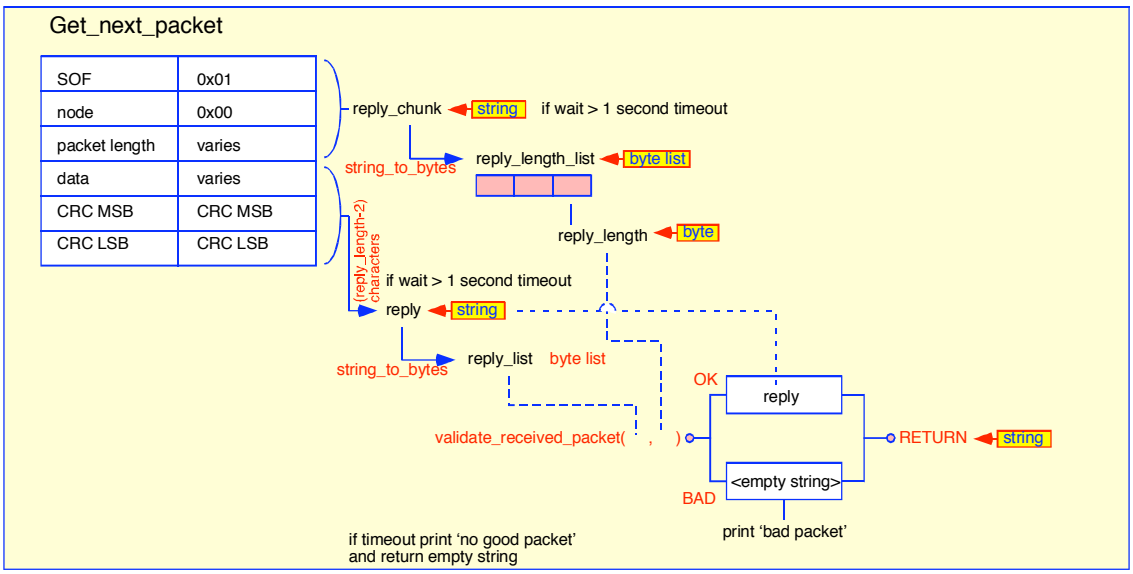
End of reply packet

SOF	0x01
node	0x00
packet length	varies
status	0x00
total IDs	2 bytes
under-run errors	2 bytes
CRC errors	2 bytes
CRC MSB	CRC MSB
CRC LSB	CRC LSB

Packet length and CRC exclude SOF byte

The core interactions with the reader happen in Get_next_packet, a routine in the module reader_functions. Bits don't appear instantly at the serial port, so you can't just read from the serial buffer as soon as you get done sending a command; instead, you need to wait until the serial port reports bytes are present. Further, you can't read a packet until you know how long it is since there is no delimiter.

Therefore, Get_next_packet reads the packet in two steps; first it waits until 3 bytes are available, so the length of the remainder of the packet can be determined. Then it reads the rest. When the packet has been read up from the serial port as a string, the contents are converted to a byte list and sent along with the length information to validate_received_packet, which checks that the packet CRC is okay. (A finite number of bad CRC's are received, even though this is a wired connection -- I don't know why.)



The remainder of this document provides a quick discussion of the various modules and routines contained therein.

class MPRGUI:

__init__:

defines the various TKinter frames and variables, handlers, etc.

note that we execute a GetReaderInfo command to get the model type
so we can set the power slider

then defines the handlers:

PCcardpot	calls Set_serial_port with PCcard, then gets model from readerinfo
USBport	calls Set_serial_port with USB, then gets model from readerinfo
getrdr	calls GetReaderInfo and puts the result in the text display widget
copyclass0	copy all the text in the Class 0 text widget
copyclass1	copy all the text in the Class 1 text widget
class0inventory	calls inventory_until_stop with EPCclass=0; reads the accumulate and antenna buttons first
class1inventory	calls inventory_until_stop with EPCclass=1; reads the accumulate and antenna buttons first
class1tagstrength	calls tagstrength with EPCclass=1;
class0tagstrength	calls tagstrength with EPCclass=0;
tagstrength	runs GetReaderInfo to get the reader model (would be nice if we had global variables!); uses the result to set the power range. Then rather ugly code in that everything is done here, no functions: -> get antenna request status, handle ALTERNATE (disallowed) -> repeat inventory until no tags are read -> run 10 inventory cycles (direct call to GetClassxInventory) -> clean up ID string format (should be a function!) -> format output with power and total reads -> step power down -> if we reach min power set totaltagsread=0 to stop -> clear the text display widget -> write the output string into it
inventory_until_stop	arguments: EPCclass, accumulate, type, antenna_index, inventory_log -> initialize -> if log is ON clear log display widget -> clear error display widget -> while NOT stop button: -> initialize the id list if NOT accumulate -> update power and antenna status -> run 10 inventory cycles: -> call GetClass[0/1]Inventory as appropriate -> increment inventory, alternate antenna if needed -> append to log display if log is ON -> if the packet is 1 byte long it's an error, parse & append -> else append new IDs to the list -> ring the bell if there are new tags or tags are lost -> clear the relevant display widget -> clean up the list for display, display it in the widget -> display the unique reads and total reads -> check the STOP button status, reset and ring 3x if STOP -> return the total number of tags read
class1verify	call VERIFY ten times, print the results -> initialize -> call GetClass1verify ten times, appending any tags to the tagidlist -> clear the display widget, clean up the strings for printing -> write the list itself, also read attempts, total tags read
parse_error_to_log	argument: error_code (a byte); returns a corresponding text message

BODY:

get reader info to flush out bad packets
set GUI instance = MPRGUI(root)
run the main loop

import serial, CRCs, packet_crunch, assemble_packets, time

set a default value of the serial port variable ser (so ser is effectively global); also define model as 'NULL' to create another global

Set_serial_port	argument: port This just sets the serial variable ser to the right device name for either the PC card emulated port or the Keyspan USB port. It is called from MPRGUI when the GUI window comes up and the user presses either the PCcard or USB button.
Start_card_get_model	Ask for the reader info until the card gives a valid packet (can take a few tries if the card was just loaded into the slot); then extract the model (MPR5000,6000, or 7000)
Append_new_IDs	Arguments: idout, newidlength, tagidlist For each id in idout, check if already in tagidlist. If not, append to both tagidlist and newidlist (which has only the new ones); return tuple (newidlist,tagidlist)
Clean_up_list	Argument: idlist format a byte list as a hex string without the 0x's; return cleaned_up_list
Get_next_packet	-> initialize -> set maximum wait = 3 seconds -> loop until either we received 3 characters OR timeout normal exit: -> read the first 3 bytes (byte 3=packet length) -> loop again until either rest of bytes are received OR timeout normal exit: -> read the buffer, convert to bytes -> Validate_Received_Packet if OK, return the last (n-3) bytes of the packet if bad, print (to terminal) and return the string 'NULL' timout: print (to terminal), return 'NULL' timeout: print (to terminal), return 'NULL'
GetReaderInfo	Call Assemble_GetReaderInfo to make a command packet Write the packet Call Get_next_packet if the reply > 55 bytes then parse it and make a human-readable string else the string is 'no info packet' and the model is 'NULL' return the reply string
Write_64bit_Class1EPC	describe later
GetClass0Inventory	arguments: power, antenna -> call Assemble_Class0Inventory with default values for singulation type -> write the command -> initialize idlist to an empty list [] -> call Get_next_packet to read the reply -> IF the reply is more than 1 byte -> convert to bytes and hex -> check the first byte [actually byte 4 of the reader packet] = status IF 0xFF then idlist = the error code ELSE call Strip_IDs_from_packet to make the real ID list if there are more packets: Get_next_packet, convert, strip IDs and append to idlist flush the input buffer return the idlist (so if the reply was 1 byte or less, idlist is an empty list)
GetClass1Inventory	arguments: power, type, antenna Pretty much the same as GetClass0Inventory except for the type spec
GetClass1verify	arguments: antenna, power Call Assemble_Class1verify, send the command Call Get_next_packet to read the response -> IF the reply is more than 1 byte -> convert to bytes and hex -> check the first byte [actually byte 4 of the reader packet] = status IF 0xFF then idlist = the error code ELSE call Strip_IDs_from_packet to make the real ID list if there are more packets: Get_next_packet, convert, strip IDs and append to idlist flush the input buffer return the idlist (so if the reply was 1 byte or less, idlist is an empty list)

followed by a bunch of text-driven interface functions no longer used...

assemble_packets.py

import CRCs, packet_crunch

Assemble_GetReaderInfo	same packet always, pre-calculated CRC
Assemble_Class0Inventory	arguments: antenna, power, singulation, filter_bits, filter NOTE: filter capability is not supported!! assemble packet, convert to bytes and calculate CRC on [packet-SOF] return packet
Assemble_Class1Inventory	arguments: antenna, power, filter_bits, filter, type Type=0 is no anti-collision, type=1 is anti-collision (quick scroll not supported) NO check for valid power range -- should add? or do check in calling routine? NOTE: filter capability is not supported!! assemble packet, convert to bytes and calaculate CRC on [packet-SOF] return packet
Assemble_Class1write	arguments: antenna, power, pointer, databyte1, databyte2 not yet tested
Assemble_Class1erase	arguments: antenna, power not yet tested
Assemble_Class1verify	arguments: antenna, power NO FILTER! note CRC doesn't agree, need to fix.

packet_crunch.py

import CRCs, serial

Strip_IDs_from_packet	argument: inputlist assumes inputlist is a packet. The first byte is the Status byte. If status is 00 (error) we return an empty list. If Status is 01, read the next byte (# of tags), build a list of IDs (nested) by extracting the bytes of the tag ID based on the first two bits. Thus assumes EPC-compliant!
Strip_longIDs_from_packet	argument: inputlist assumes inputlist is a packet. The first byte is the Status byte. If status is 00 (error) we return an empty list. If Status is 01, read the next byte (# of tags), build a list of IDs (nested) by extracting the bytes of the tag ID based on the first two bits. Thus assumes EPC-compliant! In this case, we assume each ID includes the CRC in front and kill passocde at the back, appropriate for the VERIFY command. 11/05: added error check to skip steps if the poitner gets past the end of the received data (which can happen as VERIFY just reads whatever it gets, doesn't always get all of ID)
Validate_Received_Packet	arguments: Inputlist, Recpacketlength Checks the CRC of the received packet against the calculated value. Returns 1 (TRUE) if OK, 0 (FALSE) if not. -> prepend packet length byte and the node (which is always the same) -> strip the received CRC from the last two bytes of the packet -> calculate the CRC -> if received & calc are same, return 1, else 0
String_to_bytes	arguments: input_string convert a string to a tuple of bytes
Bytes_to_hex	argument: inputlist convert an input list of numbers to a list of strings showing the values as 0xNN

CRCs.py

Class1CRC	arguments: inputlist, length, preload Inputlist is a byte list length bytes long. Preload, a hexadecimal two-byte number, is the initial state of the virtual register that implements the CRC. This routine implement the class 1 CRC algorithm, which is a bit funky and not quite the same as a 'CCITT standard' 16-bit CRC. The implementation uses bitwise operations on the numerical variable Reg to implement the shifts and adds.
MPR_CRC	arguments: inputlist, length, preload Inputlist is a byte list length bytes long. Preload, a hexadecimal two-byte number, is the initial state of the virtual register that implements the CRC. This routine implements the CCITT standard 16-bit CRC, which is identical to that used by the MPR for error checking and by the EPCglobal class 0 standard for verifying the tag EPC. In this case the routine is implemented by explicit emulation of a shift register using logical operations on a list of numerical variables register[]. (Hardly elegant but appears to be quite sufficiently fast.)

The Gen 2 class and modules are rather similar to Gen 1. Reader_functions_g2 implements the necessary inventory functions, and packet_crunch_g2 maps the grammar of G2 packets into the inventory. Class MPRGen2 creates the Gen 2 interface window and implements the control functions.

Version 6.0
Daniel M. Dobkin
Enigmatics March 28, 2006

enigmatics@batnet.com
www.enigmatic-consulting.com

Feel free to contact me with questions or enhancements to the code. I might or might not have the time or ability to help with problems but it's always nice to hear from users.