Digi TransPort® Fleet Card

User Guide

Revision history—90001245

Revision	Date	Description
B	August, 2011	Added GPS antenna information.
C	March, 2012	Updated CAN bus information.
D	July, 2012	Made minor editorial updates.
E	September, 2012	Added additional Ignition Sense Input information.
F	July, 2017	Combined the Digi TransPort Fleet Card and Fleet I/O documentation in one document, omitting Digi App Note 49, *Using the Digi TransPort Fleet Card*.

Trademarks and copyright

Disclaimers

Warranty

To view product warranty information, go to the following website:

www.digi.com/howtobuy/terms

Send comments

Documentation feedback: To provide feedback on this document, send your comments to techcomm@digi.com.

Customer support

Digi Technical Support: Digi offers multiple technical support plans and service packages to help our customers get the most out of their Digi product. For information on Technical Support plans and pricing, contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Support portal login: www.digi.com/support/eservice

Contents

About the Digi TransPort Fleet Card

The Digi TransPort Fleet Card is designed for transportation fleet applications requiring CAN/J1939, J1708, GPS, I/O and Ignition Sense interfaces.

It is fully programmable using Python to send and receive data over the CAN/J1939 and J1708 interfaces, receive GPS data and control the I/O ports.

You can also configure the Fleet Card to control the power to the TransPort router to remain powered up after the vehicle's ignition has been switched off, which allows the TransPort router to download data before powering down. The TransPort router will be switched off automatically after a configurable period of time.

Requirements

To use the functionality described in this document, you must have a Digi TransPort WR44 or WR44 R model fitted with the Fleet Card, and TransPort firmware version 5140 or later.

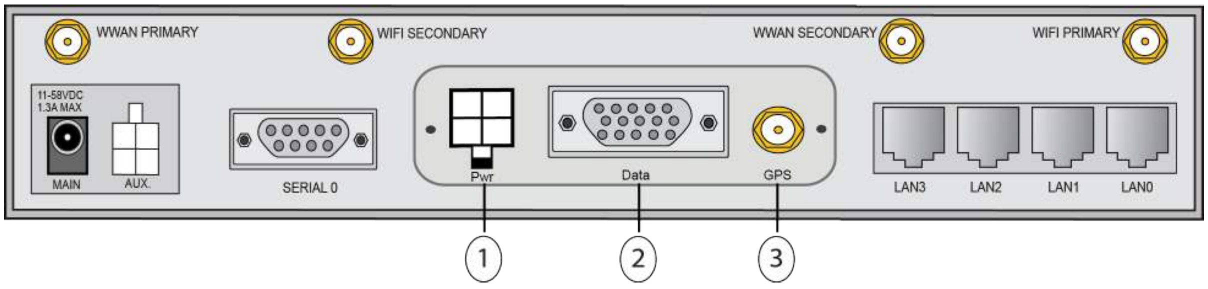
This documentation assumes that your device is set to factory default configurations. Most configuration commands are provided only if they differ from the factory default.

For successful implementation, you should have a good understanding of your product's communications technologies, and the requirements of your specific application, as well as the ability to access and navigate a Digi TransPort router, and configure it with basic routing functions.

Fleet Card features

The features of the Digi TransPort Fleet Card include:

- CAN-bus / J1939 bus
- J1708 bus
- GPS
- 3-axis accelerometer
- Local power control of the TransPort device
- 4 non-isolated digital I/O ports
- Ignition Sense Input



User accessible ports

1. Power port: This port powers the CAN-bus card.
2. Data port: This port provides access to the Fleet/J1939 interface, J1708, the 4 x Digital I/O ports, and the Ignition Sense Input.
3. GPS port: Use this SMA connector to connect the unit's GPS antenna.

   See the Hardware configuration section for pinout diagrams.

Fleet Card accessories

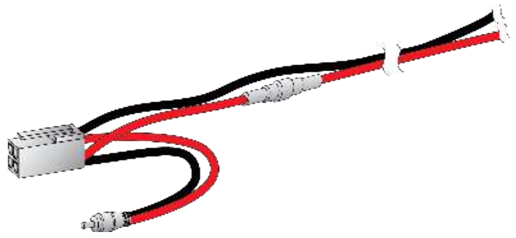
This section describes included and optional accessories for the Digi TransPort Fleet Card.

Fleet power cord (included)

This power cord is the unit's primary power source. The 4-pin connector connects to the **Pwr** port, and the locking barrel-type connector connects to the **MAIN** port.

You can order replacement cables from Digi using the following part numbers:

76000873 – Fleet Power Cable for TransPort WR44



GPS antenna (included)

The GPS antenna has an SMA connector that connects to the Fleet Card.

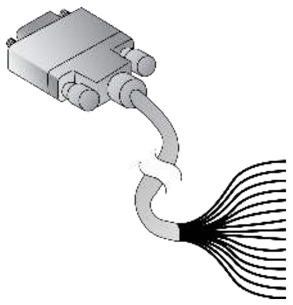
You can order a replacement antenna from Digi using the following part number:

76000842 – GPS Antenna (Magnet Mount, 1575 Mhz, 5 m cable)

Fleet telemetry cable (not included)

The Fleet telemetry cable has a D-Sub HD 15-pin connector on one end and 15 bare wires on the other end.

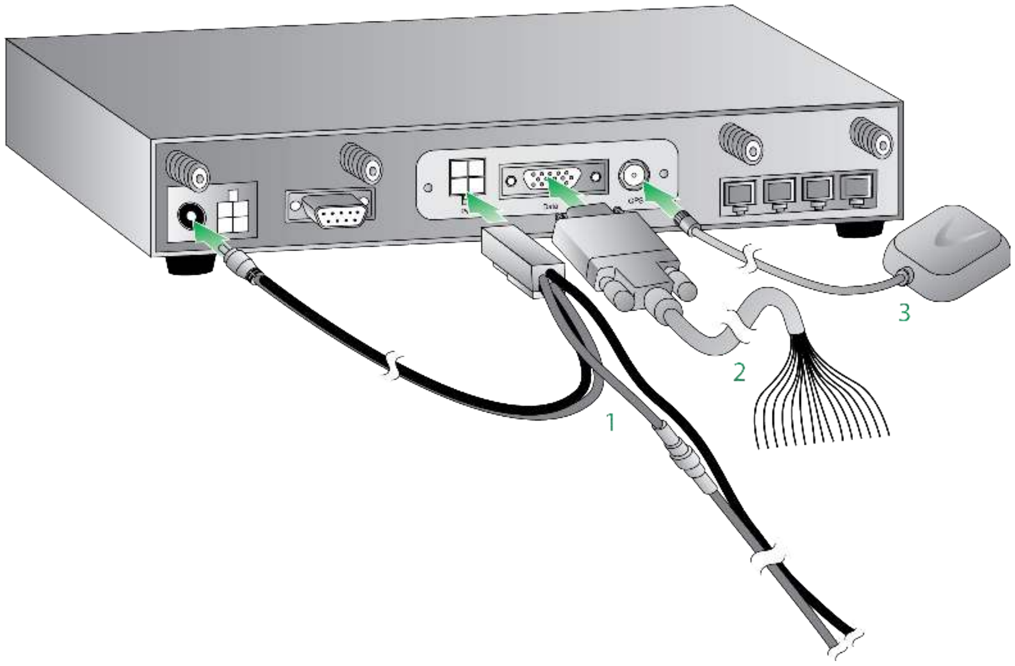
You can order a Fleet telemetry cable from Digi using the following part number: 76000872



Connecting the hardware

This section describes how to install the cables and accessories for your Digi TransPort device.

WARNING! Complete the hardware installation process prior to booting up your device. You must connect the wiring and pin placement prior to boot-up for the device to recognize the Fleet Card.



1. Connect the 4-pin connector of the Fleet Power Cord to the **Pwr** port on the unit. Connect the locking barrel-type connector to the **Main** port. Connect the bare wire ends to an appropriate power source.
2. Connect the 15-pin connector of the Fleet telemetry cable to the **Data** port on the unit. Connect the bare wire ends to the respective devices (see Pin-out diagram for a pin-out diagram of the cable).
3. Connect the GPS antenna to the device.

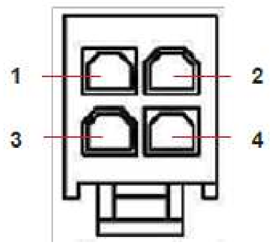
Hardware configuration

This section outlines the specifications and configuration of the interface.

Input power pin-out

The Fleet Card has the ability to locally control the power to the unit. One use of this feature is to allow the unit to remain powered up for a configured amount of time after the vehicle ignition is switched off. This allows it to download data before powering down.

The **Pwr** port uses a 4-pin Molex connector. The pin-out diagram for this port follows.



Pin	Signal
1	DC In-
2	DC Out-
3	DC Out+
4	DC In+

"DC In-" and "DC Out-" are ground returns of the input and output supply rails, respectively.

Data port pin-out

The pin-out diagram for the data port follows:

Pin	Signal
1	CAN-bus High
2	CAN-bus Low
3	J1708 Positive
4	J1708 Negative
5	J1708 Ground
6	CAN-bus Ground
7	Ignition Sense/Digital IO 0 Ground
8	Digital IO 1 Ground

Pin	Signal
9	Digital IO 2 Ground
10	Digital IO 3 Ground
11	Ignition Sense
12	Digital IO 0
13	Digital IO 1
14	Digital IO 2
15	Digital IO 3

CAN-bus/J1939 bus

CAN-bus is a vehicle bus standard that is equipped on most new vehicles and uses a differential pair of signals. The interface supports baud rates of up to 1 Mbps; you can configure the baud rates via the CLI and the Digi Python "digicanbus" module.

The interface can send and receive raw CAN messages to and from the vehicle's CAN-bus as well as 1939 messages. The interface uses the Digi Python "digicanbus" module to send and receive the messages.

J1939 is a high level vehicle bus standard that defines how communication between nodes occurs on the CAN bus.

The Fleet Card can send and receive raw CAN messages and J1939 to and from the vehicle's CAN-bus.

For more information, see Using the Digi TransPort Fleet Card.

Cable length

The CAN cable length depends on the bit rate:

Bit rate	Cable length
1 Mbps	30 m
500 kbps	100 m (normal for cars)
250 kbps	250 m (normal for trucks)

ESD protection

CAN bus lines are protected from the damage caused by ElectroStatic Discharge (ESD) and other transients with the following specification:

IEC 61000-4-2 (ESD	Level 4
IEC 61000-4-4 (EFT)	40A – 5/50 ns waveform
IEC-61000-4-5 (lightning)	8A – 8/20 us waveform
ISO 7637-1	Non-repetitive EMI surge pulse 2, 9.5 A (one 50 us pulse)
ISO 7637-3	Repetitive Fast Transient 50 A (5 x 50 us)

J1708 bus

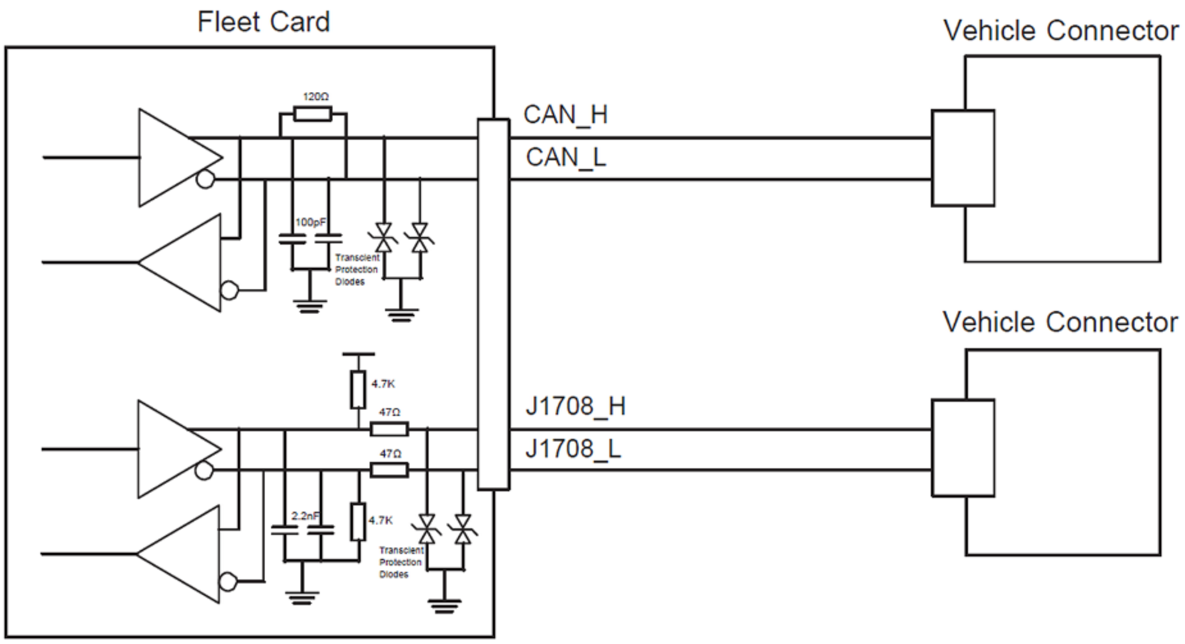
J1708 is a vehicle communications standard for heavy duty vehicles. It consists of a 2-wire (18 gauge, twisted pair) interface that operates at 9600 bits per second.

The interface can send and receive J1708 messages to and from the vehicle’s bus. The interface uses the Digi Python ”digij1708” module to send and receive the messages.

For more information, see Using the Digi TransPort Fleet Card.

CAN-bus/J1708 Rx/Tx circuit

The following diagram shows the CAN and J1708 Rx/Tx circuitry on the Fleet Card and the connection to a vehicle bus:



Digital I/O ports

There are four non-isolated digital I/O ports available. Internally, all digital I/O ports share a common ground which is the same as the vehicle's ground. The ports are internally protected against back EMF current flow. The ports are configured for input or output mode via software. Using the DIO port as input, powered drive has to be applied (either push-pull or open collector with external pull-up). Unpowered input, like passive switch to GND doesn't work. The DIO port provides open collector type driving when used as output.

Input signal

Applied input voltage to activate	+5 V to +33 V DC
Applied input voltage to deactivate	0 V to +1 V DC
Maximum input current	3 mA

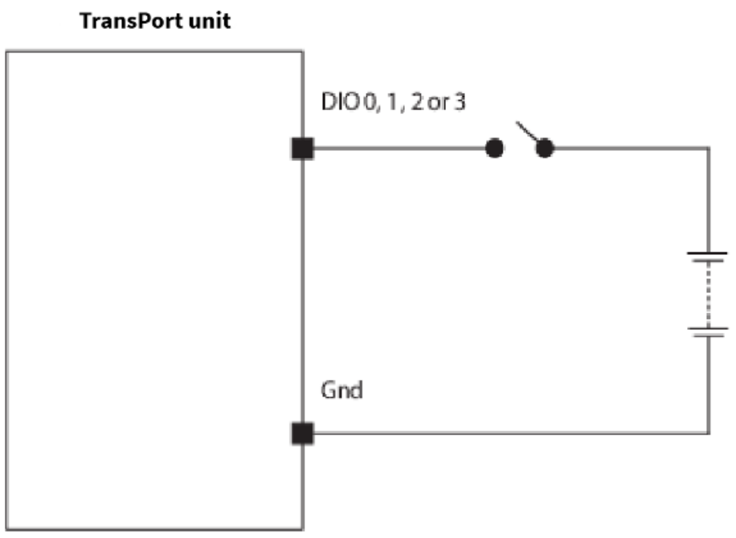
Output signal

Maximum voltage switched	+33 V DC
Maximum current switched	50 mA
Maximum leakage current	3 mA

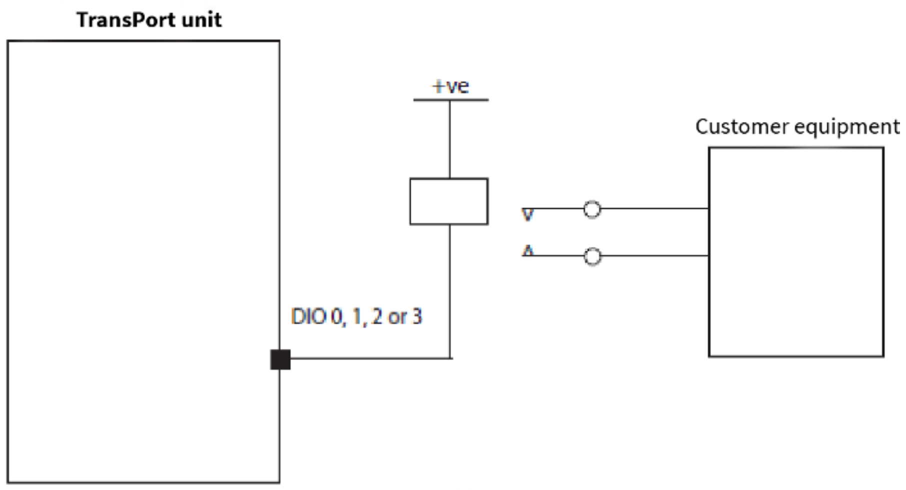
Input impedance of the DIO ports is around 10 K ohms. Overvoltage protection activates above +33 V or under -0.5 V DC. On-board 100 mA PTC fuse is used at each DIO port for current limiting in case of over/under voltage condition.

Wiring configuration

The following figures illustrate typical wiring configurations for both Input and Output applications:



Input configuration

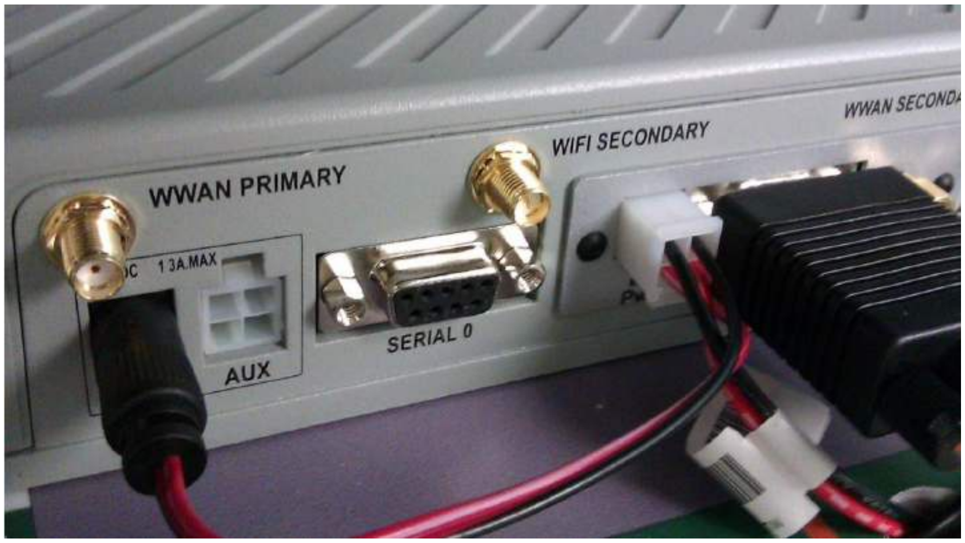


Output configuration

Power control and Ignition Sense Input

The Fleet Card can control the power to the main TransPort router by using the Ignition Sense Input. When the Fleet Card detects the Ignition Sense line going high (for example, when the engine is switched on), it will provide power to the TransPort router. When the Fleet Card detects the “Ignition Sense” signal going low (for example, when the engine is turned off), it will keep the power to the TransPort router switched on for a configurable amount of time after which the TransPort router will be switched off.

To use this feature with a WR44(R) router, use power cable 76000873.



WR44R with cable 76000873

GPS port

The Fleet Card's GPS device lets you configure the TransPort router to receive GPS messages and either process them locally or use them to forward data onto a remote device via TCP or UDP.

When the GPS data is processed locally, the longitude, latitude, altitude, number of satellites, speed, heading, and time are displayed when available.

For more information, see Using the Digi TransPort Fleet Card.

3-Axis accelerometer

You can use the 3-axis accelerometer to obtain the current forces on the X, Y, Z axes. Also, using the Digi Python “digihw_accel” module, you can set a threshold so a call-back function is called if the forces on an axis exceed the threshold.

For more information, see Using the Digi TransPort Fleet Card.

Using the Digi TransPort Fleet Card

CAN bus and J1939 commands

To configure and control the CAN bus and J1939 bus, use the command line interface (CLI).

The following are examples of useful commands.

To configure the CAN bitrate, the CLI command is as follows, where <*bitrate*> is in the range of 10000 to 1000000:

```
can 0 bitrate <bitrate>
```

The default bitrate is 250000.

To display the CAN statistics, the CLI command is as follows:

```
can 0 stats [r]
```

If the "r" parameter is entered, the CAN statistics are reset.

Use the following command to dump out CAN messages to the debug port:

```
can 0 debug
```

To access the debug, use one of thefollowing:

To send debug output to the connected Telnet session:

```
debug t
```

To send debug output to a terminal on SERIAL 0.

```
debug 0
```

Python commands for the CAN bus

The Fleet Card supports the use of the Digi Canbus Python module. You can use this module to configure the bitrate and to send and receive raw CAN messages.

To get the module, use:

```
import digicanbus
```

To create a CAN instance and configure bitrate, use:

```
can_h = digicanbus.CANHandle(<bus id>)
can_h.configure(<bitrate>)
```

To send a CAN message, use:

```
can_h.send(width, id, remote_frame, data)
```

In this example, replace the variables with the following:

`width`	11 or 29, referring to the bits in CAN identifier.

`id`	11 or 29 bit integer value.
`remote_frame`	true or false, indicating if this is a remote or data frame.
`data`	a string of 19 characters or less.

To receive a CAN message, register the receive method to a message id:

```
    can_h.register_filter (width, id, mask, recv_method, context)
```

In this example, replace the variables as follows:

`width`	11 or 29, referring to the bits in CAN identifier.
`id`	11 or 29 bit integer value.
`mask`	An id mask. It should be in the same format as the id parameter and indicates which bits in the identifier are significant for matching.
`recv_method`	The Python method to be called when a matching message is received.
`context`	Context data passed to the receive method.

The receive callback method has the following definition:

```
    def recv_method(width, id, mask, payload, context)
```

The callback method can be unregistered:

```
    can_h.unregister_filter(width, id, mask, recv_method, context)
```

You can also use the Digi Canbus Python module to create and parse J1939 PDUs.

To create a J1939 PDU:

```
    PDU = J1939_PDU()
    PDU = J1939_PDU (width, identifier, remote_frame, payload, return_arg)
```

To create a CAN message from a J1939 PDU:

```
    can_msg = PDU.CANMsgTuple()
```

CAN bus Python example

```
## Import the CAN module
from digicanbus import *
import struct, time, sys

speed = 125000
if len(sys.argv) >= 2:
  speed = int(sys.argv[1])

## The digicanbus module has CANHandle(). A function that returns
## a handle to the current CAN bus.

## Specify the CAN bus number.
print "Getting handle to CAN bus 0"
handle = CANHandle(0)

## First we configure the bus to the speed specified
print "Configuring for %d bps" %speed
handle.configure(speed)

## We create a simple callback function to use with the CAN filters.
## The callback must have the following parameters defined:
## width: specifies either 11 or 29 bit message
## identifier: The identifier that was matched
## remote_frame: Boolean indicating a remote frame (RTR)
## payload: 0-8 bytes of payload for the message
## return_arg: Argument specified when creating the filter

## We will use the return_arg parameter to determine which filter
## triggered the callback.

def callback_1(width, identifier, remote_frame, payload, return_arg):
  print "\ncallback_1 function was called"
  print '11 or 29 bit: ', width, ' bit'
  print 'Identifier matched: ', identifier
  print 'Is remote frame: ', remote_frame
  print 'Payload: ', struct.unpack('%dB'%len(payload), payload)
  print 'Return arg: ', return_arg

## We create a tuple, which contains all the information needed for
## the filter
## Width: 11 or 29 bit message
## Identifier: Which CAN identifier will be selected
## Mask: Which bits of the identifier matter when matching
## callback function: The function will be called when something
##   is matched
## return_arg: Argument passed to the call back when

## Note: Multiple filters can use the same callback function

## Below we are exploring different scenarios with the filters.

print "Defining filters:"
## Filter 1 will only trigger on messages with the 0x700 identifier.
## This is done by saying that the bits 0x700 must be on, and all
## bits will be measured in the mask (0x7FF).
```

```
filter_1 = (11, 0x700, 0x7FF, callback_1, 'filter_1')
print "Filter 1: ", filter_1

## Filter 2 will trigger on messages between 0x700 and 0x7FF.
## This is done by saying that the bits 0x700 must be on, but only
## the 0x700 bits will be measured in the mask.

filter_2 = (11, 0x700, 0x700, callback_1, 'filter_2') print "Filter 2: ", filter_
2

## Filter 3 will trigger on all 29 bit messages.
## This is done by setting the width to 29, and using values 0x0 for
## identifier and mask.

filter_3 = (29, 0x0, 0x0, callback_1, 'filter_3')
print "Filter 3: ", filter_3

## Register the filters on the CAN bus
print "Registering filters..."
handle.register_filter(*filter_1)
handle.register_filter(*filter_2)
handle.register_filter(*filter_3)

counter = 0
print "Hit enter to send a CAN message, type 'quit' to exit"
while raw_input().lower() != 'quit':
  counter += 1
  msg = (11, counter % 0x500, False, str(counter % 99999))
  handle.send(*msg)

## Unregister the filters created using the stored tuples
print "Unregistering filters"
handle.unregister_filter(*filter_1)
handle.unregister_filter(*filter_2)
handle.unregister_filter(*filter_3)
```

J1939 bus Python example

```
## Import the CAN module
from digicanbus import *
import struct, time, sys

baud = 125000 if len(sys.argv) >= 2:
baud = int(sys.argv[1])

## The digicanbus module has CANHandle(). A function that returns a handle
## to the current CAN bus.

## Specify the CAN bus number.
print "Getting handle to CAN bus 0"
handle = CANHandle(0)

## Configures the CAN bus to a specific bps and starts it. This must be
## called at least once.
print "Configuring the bus to %d bps" % baud
handle.configure(baud)

## We create a function to be called when a J1939 message that is matched
## will be passed to.

def callback_1(width, identifier, remote_frame, payload, return_arg):
    PDU = J1939_PDU(width, identifier, remote_frame, payload)
    print "====PDU received===="
    for opt in ['DA', 'DP', 'EDP', 'GE', 'PF', 'PGN',
                 'PS', 'SA', 'priority', 'payload']:
        print opt + " = " + str(PDU.__getattribute__(opt))

    print "Converting to a raw can message, sending it over the CAN bus"
    raw_msg = PDU.CANMsgTuple()
    print "%s %s %s %s" %(raw_msg[0], hex(raw_msg[1]), raw_msg[2],
raw_msg[3])
    try:
        handle.end(*raw_msg)
    except Exception, e:
        print e
    else:
        print "Message succesfully sent"

## We create a filter that will trigger on 29 bit messages
filter_1 = (29, 0x0, 0x0, callback_1, 'filter_1')
print "Filter 1: ", filter_1

## Register the filters on the CAN bus
print "Registering filters..."
handle.register_filter(*filter_1)
counter = 0
print "Hit enter to send J1939_PDU, type 'quit' to exit"
while raw_input().lower() != 'quit':
counter += 1
    P = J1939_PDU()
    P.DA = 0x76
    P.PGN = 0xF001
    P.payload = 'msg' + str(counter)
    can_msg = P.CANMsgTuple()
    handle.send(*can_msg)

## Unregister the filters created using the stored tuples print "Unregistering
filters"
handle.unregister_filter(*filter_1)
```

J1708 bus commands

This section provides examples for setting up the J1708 communication with the vehicle's bus.

The Fleet Card supports the use of the Digi J1708 Python module. It can be used to configure the bus and to send and receive J1708 messages.

To get the module use:

```
import digij1708
```

To create and configure a J1708 instance, use the following:

```
j1708_h = digij1708.J1708Handle(<bus id>)0
j1708_h.configure(<options>)
```

The valid options are:

`mid=nJ1708`	This is the AE J1708 message identification character. No two devices on the bus should share a message identifier, per specification. '*n*' must be between 0 and 255. If the **mid** option is left unconfigured, sending capability is disabled.

To send a J1708 message, use the following:

```
j1708_h.send(priority, data)
```

In this example, replace the variables as follows:

`priority`	1 – 8, as defined in the SAE J1708 specification.
`data`	a string of 19 characters or less.

To receive a J1708 message, the receive method must be registered:

```
j1708_h.register_filter(recv_method, context)
```

In this example, replace the variables as follows:

`recv_method`	1 – 8, as defined in the SAE J1708 specification.
`context`	Context data passed to the receive method.

The Python method to be called when a matching message is received.

The receive callback function has the following definition:

```
def j1708_recv_function(mid, payload, context)
```

A callback function can be unregistered:

```
j1708_h.unregister_filter(recv_method, context)
```

J1708 Python example

```
## Import the J1708 python API
import digij1708
import struct

## Create a handle to the J1708 bus through the class J1708Handle()
## The class takes 1 parameter, the bus number to get a handle to.
print "Getting a handle to the J1708 Bus"
handle = digij1708.J1708Handle(0)

## Configure the bus to a particular mid. in this case 0x48
handle.configure(0x48)

##Create a callback function.   This function requires three parameters:
##  mid, payload, and arg

## mid: The message ID
## payload: The payload of the message
## arg:        An arbitrary parameter defined when setting the callback.

def callback_1(mid, payload, arg):
  print "\nMid: ", mid
  print "Payload: ", payload
  print "Arg: ", arg

## Register the callback on the J1708 bus
handle.register_callback(callback_1, 'foo')
print "Type 'quit' to exit, or type a message in to send then hit enter:\n"
while 1:
  input = raw_input()
  if input.lower() in ['q', 'qu', 'qui', 'quit']:
    break

input = input.strip()

## If no input, create a fake message
if len(input) == 0:
  msg = struct.pack('=2B', 45, 12)

## If input, take up to 19 characters of it
else:
  if len(input) > 19:
    msg = input[:19]
  else:
    msg = input

## Send the message at priority 1
try:
  handle.send(1, msg)
except Exception, e:
  print e
## Unregister the call back, must use exact same input as the register call
print "Unregistering callback"
handle.unregister_callback(callback_1, 'foo')
```

GPS commands

This section provides examples for setting up the GPS communication.

The TransPort router supports the Digi hardware Python module, digihw.

To get the module, use:

```
    import digihw
```

To get the GPS data, use the following command:

```
    gpsData = digihw.gps_location()
```

In this example:

`gps data`	A tuple containing latitude, longitude, altitude and timestamp.

GPS Python example

```
## The returned values from the gps_location call are NMEA parsed into a
## tuple: (latitude, longitude, altitude, timestamp)

## The sample returned is the latest received from the GPS device.      The
## timestamp is assigned to it when it was successfully parsed.
## In cases where there are no sample available, an exception is raised.

"""\
    GPS Location API Sample
    This example reads and displays all the values from the device's
    integrated GPS each 5 seconds, using the GPS Location API.

    Displays Latitude, Longitude, Altitude, Timestamp in GMT/GPS time
"""
# imports
import sys
import os
import time
import digihw  ## From the digi embedded library, import the parsed
gps_location

# variables is_reading = True

def get_formmated_value(value):
    """
        Returns the given value formatted in degrees, minutes and seconds.
    """
    working_value = value
    if working_value < 0:
        working_value = working_value * -1
    deg = working_value;
    gpsdeg = int(deg)
    remainder = deg - (gpsdeg * 1.0)
    gpsmin = remainder * 60.0
    remainder2 = gpsmin - int(gpsmin)*1.0
    gpsseg = int(remainder2*60.0)

final_value = "%sdeg %smin %ssec" %(gpsdeg, int(gpsmin), gpsseg)
```

```
return final_value
def get_latitude_hemisphere(latitude):
"""
Returns the hemisphere depending on the latitude (S or N)
"""
if latitude < 0:
return "S"
return "N"
def get_longitude_hemisphere(longitude):
"""
Returns the hemisphere depending on the longitude (W or E)
"""
if longitude < 0:
return "W"
return "E"

# Read GPS information every 5 seconds while is_reading:
try:
print "Reading GPS data...\r\n" gps_data = digihw.gps_location()
latitude, longitude, altitude, timestamp = gps_data
print "Latitude        : %s %s" %(get_formmated_value(latitude),
get_latitude_hemisphere(latitude))
print "Longitude : %s %s" %(get_formmated_value(longitude),
get_longitude_hemisphere(longitude))
print "Altitude : %d meters" %altitude
print "Date    : % s" %time.ctime(timestamp)
# Wait 1 seconds
print "Please, wait 1 seconds...\r\n"
time.sleep(1)
except:
print "Couldn't read GPS data. You need to get a better GPS signal.\n
Please, ensure the GPS antenna is correctly connected and \
has an open view of the sky.\r\n"

# Wait 20 seconds
print "Please, wait 20 seconds...\r\n\r\n"
time.sleep(20)
```

Accelerometer commands

The Fleet Card accelerometer can detect movement in all directions.

To take a sample of the current forces, use the following CLI command:

```
accel 0 show [num of samples]
```

If a threshold is exceeded, the TransPort router displays the previous 33 samples on the debug port.

To set the threshold for the accelerometer, use the following CLI command:

```
accel 0 set <threshold>
```

In this example, replace the threshold variable as follows:

```
<threshold> = (0.00-15.99)*64)
```

For example, to set a threshold of 1.5G, you would enter 96 (= 1.5 x 64).

Python commands for the accelerometer

The Fleet Card supports the use of the Digi Accelerometer Python module. You can use this Python module to set thresholds and read the current forces being detected.

To get the module, use:

```
import digihw
```

To take a sample of the current forces, use the following:

```
accel = digihw.accelerometer()
accel.sample()
```

This method returns a 3-tuple (x, y, z) representing the g-forces measured in the X, Y and Z axes.

You can configure the Fleet Card to call a callback method when a certain threshold is exceeded:

```
accel = digihw.accelerometer()
accel.sample()
```

This method returns a 3-tuple (x, y, z) representing the g-forces measured in the X, Y and Z axes.

You can configure the Fleet Card to call a callback method when a certain threshold is exceeded:

```
accel = digihw.accelerometer()
accel.register_threshold(threshold, method, context)
```

In this example, replace the variables as follows:

`threshold`	G-force threshold on any axis
`method`	Callback method
`context`	Context data passed to the callback method

The callback method has the following definition:

```
def callback_method(sample, context)
```

In this example, replace the variables as follows:

`sample`	3-tuple (x, y, z) representing the g-forces measured in the X, Y and Z axes

`context`	Context data passed to the callback method

Accelerometer Python example

```
#
# Sets the callback threshold and displays on which axes the
# threshold is exceeded.
#
import digihw
import time
import sys

AXIS_X = 0
AXIS_Y = 1
AXIS_Z = 2

if len(sys.argv) != 2:
    print "Usage: python accel.py <threshold>"
    sys.exit(-1)
threshold_g = float(sys.argv[1])

def accel_callback(sample, context):
    x_g = sample[AXIS_X]
    y_g = sample[AXIS_Y]
    z_g = sample[AXIS_Z]

    if x_g >= threshold_g:
        print("Threshold exceeded on X axis (%fG)" % x_g)

    if y_g >= threshold_g:
        print("Threshold exceeded on Y axis (%fG)" % y_g)

    if z_g >= threshold_g:
        print("Threshold exceeded on Z axis (%fG)" % z_g)


print "Digi TransPort Accelerometer Python example"

# Register the callback function and wait for 10 seconds
print ("Threshold set to %fG" % threshold_g)
accel = digihw.accelerometer()
accel.register_threshold(threshold_g, accel_callback, 0)

time.sleep(10);

print "Complete"
```

Ignition Sense Input commands

This signal controls the power up/down of the Fleet Card and may optionally also control the host, as required. The purpose of this signal is to allow the router to be permanently connected to a +12 or +24 V vehicle supply, but the time that the router is operational is governed by this input. Assuming the presence of the vehicle supply in the Fleet Card's power input, then when power is supplied to this input (typically in the range +12 V to +24 V with some margin either way) the Fleet Card will power up. If the Fleet Card power cable's "Host Extension" is looped back into the power socket of the host, the Fleet Card will now also power up the host. When the "Ignition" signal is disconnected from the +12 V/+24 V supply (perhaps because the driver turned off the vehicle engine), a timer on the Fleet Card starts a countdown to the time when both the Fleet Card and the host (if powered via the Fleet Card) the host will power down. This countdown period is software configurable. The purpose of this is to allow the router enough time to transmit data such as journey statistics and present location before the system shuts down.

Users can select one of two configurations:

- The router/Fleet Card is running while the engine is on and will continue to run for a preset time afterwards.
- The router/Fleet Card is running only when power is applied (all power control is either manually managed or is controlled by some other system separate from the router).

Two wiring arrangements are available for these scenarios:

1. **Fleet-card controlled power**

   Wire the power input to the Fleet Card directly into a permanent battery (+12 V or +24 V) supply. Connecting directly to the battery is best, using thick wire of as short a length as possible to minimize losses in the cable. Connect the Fleet Card ignition signal to the vehicle ignition. This is the second position on most key switches, and should be the position for the engine to be running normally. Connect the power output from the Fleet Card (the Host Extension) into the power input of the host router. Turn on the vehicle ignition and when the router has booted, configure the "Router Stay Alive" time as required.

2. **Externally controlled power**

   Wire the power to the Fleet Card directly to the power source. Wire the Fleet Card ignition to the same power source. if this step is not done, the Fleet Card will not operate. Wire the Host router directly to the same power source. Do NOT use the Host Extension power connection coming out of the Fleet Card for this as the direct connection reduces losses in the system.

Commands for Ignition Sense Input

To configure the delay between the "Ignition Sense" signal going low and the TransPort router being switched off, use the following command:

```
fleet 0 holdon <secs>
```

If this feature is not required, the TransPort router should be powered by a separate power cable.

The status of the ignition signal can be retrieved using the following CLI command:

```
fleet ignition
```

Digital I/O port commands

The ports are configured for input or output mode via software. The CLI command is as follows:

```
    fleet gpio <port> <in|out>
```

To enable or disable the GPIO output port, the CLI command is as follows:

```
    fleet gpio <port> <on|off>
```

To retrieve the status of the ports, the CLI command is as follows:

```
    fleet gpio
```

Python commands for the digital I/O ports

The Fleet Card supports the use of the Digi hardware Python module, digihw. You can use this Python module to manipulate the digital I/O ports.

To get the module, use:

```
    import digihw


    digihw.gpio_set_value(<port>,<0|1>)
    digihw.gpio_set_input(<port>)
    digihw.gpio_get_value(<port>)
```

Digital I/O Python example

```
#
# Sets port 0 level to 1. It then sets port 1 to be an input
# and waits for the level to go to 0. When this has happened,
# it sets port 0 back to 0.
#

import digihw
import time

PORT_0 = 0
PORT_1 = 1

print "Digi TransPort Digital I/O Python example"

digihw.gpio_set_value(PORT_0, 1)

digihw.gpio_set_input(PORT_1)

print “Waiting for Port 1 to go low” while digihw.gpio_get_value(PORT_1) != 0:
    time.sleep(1)

digihw.gpio_set_value(PORT_0, 0)

print "Complete"
```