OEM7
NovAtel API
User Manual

OEM7 NovAtel API User Manual

Publication Number: D100407

Revision Level: v2

Revision Date: May 2025

Firmware Versions: 7.09.05 / OM7MR0905RN0000

Proprietary notice

Table of Contents

Chapter 8 Lua logs

Chapter 9 Using Lua to access I/O devices connected to the receiver

Customer Support

If you have any questions or comments regarding your OEM7 product, contact NovAtel Customer Service.

Log a support request with NovAtel Customer Support using one of the following methods:

Log a case and search knowledge:

Website: novatel.com/support

Log a case, search knowledge and view your case history: (login access required)

Web Portal: shop.novatel.com/novatelstore/s/login/

E-mail:

support.novatel@hexagon.com

Telephone:

U.S. and Canada: 1-800-NOVATEL (1-800-668-2835)

International: +1-403-295-4900

Lua language

Contact Lua by visiting their web site: www.lua.org/contact.html

Additional NovAtel documentation

To view the complete OEM7 suite of user documentation, go to the NovAtel OEM7 Receiver Documentation Portal at: docs.novatel.com/OEM7

Chapter 1 Overview

NovAtel API is used to develop specialized applications using the Lua programming language to further extend the functionality of the OEM7 family receiver. Lua scripts created by customers run alongside the core receiver firmware using an embedded Lua script interpreter. The scripts can interact with the core firmware by sending commands to the receiver and retrieving logs for processing.

1.1 Features

NovAtel API provides the following features:

- Powerful scripting capability using Lua, a popular scripting language for embedded applications
- Dedicated sockets allowing Lua scripts to directly send commands to and receive logs from the receiver firmware
- Special Tunneling Ports provide access to physical ports on the receiver

Lua scripts can be used to:

- Create customized logs to be sent out a communication port
- Intercept the command stream for creating and interpreting custom commands

1.2 Materials provided – NovAtel API

NovAtel API supports:

- The utility programs **TOSREC**, **DATABLK** and **MKISOFS**, which are utilities used to create an ISO9660 file system image and format that image for use with a NovAtel receiver
- Examples
- Release Notes. The Release Notes should be read carefully to understand changes made since the last release.
- ZeroBrane Lua Integrated Development Environment (IDE)

1.3 Requirements to use NovAtel API

In addition to the items provided with NovAtel API, the following items are required to create and run a Lua application on an OEM7 family receiver:

- A PC is required to write the application files and run the utilities that package the Lua scripts for use on the receiver
- The PC requires a connection to the receiver (serial port, USB, Ethernet, etc.) to load the application onto the receiver
- OEM7 family receiver running OM7MR0500RN0000 (7.05.00) firmware or higher, or a PwrPak7 receiver running EP7PR0504RN0000 (7.05.04) firmware or higher, loaded with a user application enabled software model
- In addition to this manual, the NovAtel OEM7 Receiver User Documentation Portal (docs.novatel.com/OEM7) is an online reference for other OEM7 receiver information. PDF versions of manuals are available for download from this location as well.

The Lua interpreter is present on the receiver, so no compiler or specialized development tools are needed to create an application. However, a Lua development environment called ZeroBrane is provided within NovAtel API.

The OEM7 receiver contains version 5.3.4 of the Lua interpreter. Be sure that any offline development is done using this version of the Lua interpreter.

1.4 Compatibility with applications built for OEM6 receivers

Existing applications, created for the OEM6 family of receivers, are not compatible with the OEM7 family receivers and will need to be redesigned and written as a Lua script. Some functionality that was available in the OEM6 Application Programming Interface will not be available for Lua scripts. Consult the release notes for functional compatibility with the OEM6 Application Programming Interface.

Chapter 2 Concepts

“Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.” (quoted from www.lua.org/about.html)

A Lua interpreter has been embedded into the OEM7 firmware and can be used to add additional functionality to the OEM7 receiver. Using Lua's native socket module, a script can connect to a virtual NovAtel port (SCOM) to communicate with the receiver using the standard NovAtel commands and logs.

2.1 Required firmware model

To use the Lua interpreter, a model supporting user applications is required. This is indicated by a trailing "A" in the model name and can be confirmed using the **MODELFEATURES** log, which must show that the user application (API) is AUTHORIZED.

Here is an example of a model with a trailing "A" that supports the user application:

```
log version
<OK
[COM2]<VERSION COM2 0 92.5 UNKNOWN 0 2352.778 02444020 3681 14581
<     3
<          GPSCARD "DDNRNNCBNA" "BMHR16370009M" "OEM7700-1.00" "OM7MR0400AN0001"
"OM7BR0000RB0000" "2018/Jan/09" "07:58:45"
<          OEM7FPGA "" "" "" "OMV070001RN0000" "" "" ""
<          DB_LUA_SCRIPTS "SCRIPTS" "Block1" "" "SAMPLE1" "" "2018/Jan/22"
"14:19:44"
[COM2]


log modelfeatures
<OK
[COM2]<MODELFEATURES COM2 0 90.5 UNKNOWN 0 2358.402 02444020 141a 14581
<     20
<          20HZ MAX_MSR_RATE
<          20HZ MAX_POS_RATE
<          SINGLE ANTENNA
<          AUTHORIZED MEAS_OUTPUT
<          AUTHORIZED DGPS_TX
<          AUTHORIZED RTK_TX
<          AUTHORIZED RTK_FLOAT
<          AUTHORIZED RTK_FIXED
<          AUTHORIZED PPP
<          AUTHORIZED LOW_END_POSITIONING
<          AUTHORIZED RAIM
<          AUTHORIZED API
<          AUTHORIZED NTRIP
<          UNAUTHORIZED IMU
<          UNAUTHORIZED INS
<          UNAUTHORIZED ALIGN_HEADING
<          UNAUTHORIZED ALIGN_RELATIVE_POS
<          UNAUTHORIZED INTERFERENCE_MITIGATION
<          UNAUTHORIZED RTKASSIST
<          UNAUTHORIZED SCINTILLATION
[COM2]
```

2.2 Getting started with Lua

To quickly start the Lua interpreter, connect to any port of the OEM7 receiver and send the command **LUA PROMPT**. This will do two things:

1. Change the Interface Mode of the port that received the command to LUA. In the example below, that's COM1.
2. Start the Lua interpreter

Interface mode **LUA** establishes a connection between the COM port and the Lua interpreter's stdin, stdout and stderr. This connection allows commands to be typed directly to the Lua interpreter (stdin) and the output from print statements is sent to the COM port. For more information, see *Loading and running the application* on page 18.

From there, Lua commands can be entered as shown below.

```
lua prompt
<OK
[COM1]Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
>
>
> print("Hello World")
Hello World
>
> Var1 = 1
> Var2 = 2
> print(Var1+Var2)
3
>
```

NovAtel Connect's Console Window cannot be used for this purpose because it depends on the port remaining in the NOVATEL Interface Mode. The terminal emulators TeraTerm or Hyperterm can be used:
https://teratermproject.github.io/index-en.html
https://hyperterminal-private-edition-htpe.en.softonic.com/download

For details on the Lua language, reference manuals can be found at www.lua.org. The OEM7 receiver uses Lua version 5.3.4.

2.3 Using SCOM ports

Lua interacts with the rest of the OEM7 receiver using SCOM ports. SCOM ports are similar to ICOM ports, except they have fixed socket port numbers and use only UDP.

The Lua socket library is compiled into the OEM7 receiver and is used to communicate with the SCOM ports. Details on the Lua socket library can be found here: https://github.com/diegonehab/luasocket

These are the steps to setup an SCOM connection in Lua.

1. Use the Lua **`require`** function to initialize the socket library.

```
SocketLib = require("socket")
```

2. Use the socket library to get an instance of a UDP object.

```
SocketSCOM1 = SocketLib.udp()
```

UDP communication is used to improve performance. Although the UDP protocol is normally considered "unreliable" over Ethernet, it is very reliable and efficient for connections on a local host.

3. Setup the socket.
   - Since the Lua interpreter is running on the OEM7 receiver, use the localhost (127.0.0.1) IP address.
   - Use the NovAtel-added **`scom`** module to convert from the SCOM number to the socket port number.
   - Wrap the calls with the **`assert`** function to check for errors.

     ```
     assert(SocketSCOM1:setsockname("*",0))
     assert(SocketSCOM1:setpeername("127.0.0.1",scom.GetSCOMPort(1)))
     assert(SocketSCOM1:settimeout(3))
     ```

4. The socket is now ready to send and receive data. Use the **`:send()`** method to issue a command to the receiver through the SCOM socket. Use the **`:receive()`** method to retrieve the receiver's response to the command and also to receive the requested logs or other data from the receiver.

This example shows how to use the socket object created above to collect a VERSIONA log:

```
SocketSCOM1:send("log versiona\r")

while(true) do
  Buffer = SocketSCOM1:receive()
  if Buffer == nil then
    print("... timed out")
    break
  end
  print("> ", Buffer)
end
```

2.4 Sending data out a receiver port using SEND or SENDHEX

The Lua interpreter uses the standard NovAtel commands and logs and therefore does not have special access to the ports on the receiver. However, the **SEND** command and **SENDHEX** command can be used to output data on any desired receiver port. This is the method to use when other NovAtel logs are coming out of the port. Data sent using the SEND or SENDHEX commands will not corrupt the other logs on the port.

Here is a simple example of how to do this:

```
SocketLib = require("socket")
SocketSCOM1 = SocketLib.udp()

assert(SocketSCOM1:setsockname("*",0))
assert(SocketSCOM1:setpeername("127.0.0.1",scom.GetSCOMPort(1)))
assert(SocketSCOM1:settimeout(3))
SocketSCOM1:send("send com2 \"Hello World\n\"\r")
SocketSCOM1:send("sendhex com2 12 48656C6C6F20576F726C640A\r")
```

In this example, the script opens up SCOM1 and then uses the **SEND** command to send "Hello World\n" as a string and then uses the **SENDHEX** command to send the equivalent hex data. When the script is run, two instances of "Hello World\n" are output on COM2:

```
[COM2]Hello World

Hello World
```

Note the use of backslashes to escape special characters to form a string within a string.

2.5 Using a tunnel to take over a port

An alternative to the SEND / SENDHEX commands is to establish a tunnel between an SCOM port and an external port. In this configuration, all data sent into the SCOM will be output on the external port and all data on the external port will be sent out the SCOM.

Below is a simple example, which sets up an echo on COM2. For a more extensive example of taking over a port, see the *intercept.lua* script within the sample scripts folder of the development kit.

```
SocketLib = require("socket")

-- Use SCOM1 for commands and logs
local SocketSCOM1 = SocketLib.udp()
-- Use SCOM2 for the tunnel
local SocketSCOM2 = SocketLib.udp()

-- Setup the sockets
TargetIP = "127.0.0.1"

assert(SocketSCOM1:setsockname("*",0))
assert(SocketSCOM1:setpeername(TargetIP,scom.GetSCOMPort(1)))
assert(SocketSCOM1:settimeout(3))

assert(SocketSCOM2:setsockname("*",0))
assert(SocketSCOM2:setpeername(TargetIP,scom.GetSCOMPort(2)))
-- No time out on SCOM2

-- Create function to send a command and wait for a prompt
-- Returns the prompt on success, nil on failure
function WaitForPrompt(SocketSCOM_)
  while true do
    local Buffer = SocketSCOM_:receive()
    if Buffer == nil then
      print("Timed out")
      return nil
    end

    local Start,Stop,Prompt = Buffer:find("(%[SCOM%d%])")

    if Prompt ~= nil then
      print("Prompt Received: ",Prompt)
      return Prompt
    end
  end

  return nil
end
```

```
-- Send a one-byte packet to SCOM2 so that it knows the IP address of the
machine
-- running the Lua script
SocketSCOM2:send("\r")

-- Setup the tunnel on the SCOM2 side
SocketSCOM1:send("interfacemode scom2 tcom2 none\r")
assert(WaitForPrompt(SocketSCOM1))

-- Setup the tunnel on the COM2 side
SocketSCOM1:send("interfacemode com2 tscom2 none\r")
assert(WaitForPrompt(SocketSCOM1))
SocketLib.sleep(1)

-- Setup an echo loop
-- This will have the effect that if the user enters characters
-- on COM2, they will be echoed back
while true do
  -- Receive characters from SCOM2
  local Buffer = SocketSCOM2:receive(1)
  print ("Buffer: ",Buffer)
  -- Echo those characters back to SCOM2
  SocketSCOM2:send(Buffer)
end
```

SCOM and connectionless UDP

The UDP communication used on the SCOM ports is connectionless, which means that the SCOM side does not know the IP address of the Lua interpreter until the Lua interpreter has sent a byte to the SCOM. Therefore, no data will be received on an SCOM until a byte has been sent to it.

That's why in the example above, a one-byte packet is sent to SCOM2 before attempting to receive on the socket.

Chapter 3 Learning Lua

3.1 Online documentation

An introduction to programing in Lua is available on lua.org here: www.lua.org/pil/contents.html. This free online version is based on Lua version 5.0, but it remains a good starting point for developers new to the language.

Newer versions of the programming guide are available for purchase.

3.2 Creating a custom NovAtel style log

The Lua string library can be used to parse NovAtel ASCII logs and create new custom logs. The example below shows how to do that. Note the following:

- The **`string.find`** function is used to split the TIMEA log into its header and data.
- The **`string.gmatch`** function is then used to split up the individual comma-separated data fields. The data fields are then stored into table, which can be used as required.
- The **`string.format`** function is used to format a new log.

A tutorial on the Lua String Library can be found here: www.lua.org/pil/20.html.

The full script, as well as the required crc32.lua module is available in the Lua Dev Kit.

```
--********************************************************************************
-- Parse a string, looking for a TIMEA log
-- Inputs:
--   Buffer_ String containing input data

-- Returns:
--   nil if no TIMEA log is found
--   A table representing the data of a TIMEA log if a log is found
--********************************************************************************
function ParseTIMEA(Buffer_)

  -- Search for a TIMEA log.
  -- string.find returns the start and stop index as well as any strings that
are "captured" within the parentheses
  local FindTIMEAStart
  local FindTIMEAStop
  local TIMEAHeader
local TIMEAData

  FindTIMEAStart,FindTIMEAStop, TIMEAHeader,TIMEAData
    = Buffer_:find("#(TIMEA[^;]*;)([^%*]*%*).-\n")

  if FindTIMEAStart ~= nil then
  -- Found a TIMEA log

    -- split the header into its elements
    local HeaderIter = TIMEAHeader:gmatch("([^,]-)[,%;]")
    HeaderData = {}
    HeaderData['Message'] = HeaderIter()
    HeaderData['Port'] = HeaderIter()
    HeaderData['Sequence'] = HeaderIter()
```

```
    HeaderData['IdleTime'] = HeaderIter()
    HeaderData['TimeStatus'] = HeaderIter()
    HeaderData['Week'] = HeaderIter()
    HeaderData['Second'] = HeaderIter()
    HeaderData['ReceiverStatus'] = HeaderIter()
    HeaderData['Reserved'] = HeaderIter()
    HeaderData['ReceiverSWVersion'] = HeaderIter()

    -- Split the data into its elements
    -- gmatch returns an iterator function that can be called successively to
get
    -- the next string matching the pattern.
    local DataIter = TIMEAData:gmatch("([^,]-)[,%*]")

    -- Create a table for the Time Data and assign the data fields into that
table
    TimeData = {}
    TimeData['Header'] = HeaderData
    TimeData['ClockStatus'] = DataIter()
    TimeData['Offset'] = DataIter()
    TimeData['OffsetStd'] = DataIter()
    TimeData['UTCOffset'] = DataIter()
    TimeData['UTCYear'] = DataIter()
    TimeData['UTCMonth'] = DataIter()
    TimeData['UTCDay'] = DataIter()
    TimeData['UTCHour'] = DataIter()
    TimeData['UTCMinute'] = DataIter()
    TimeData['UTCMillisecond'] = DataIter()
    TimeData['UTCStatus'] = DataIter()

    return TimeData
  end
  -- NOTE:  There is an implicit return of nil for Lua functions
 --         that do not otherwise return a value
end

--*************************************************************************
-- Create a custom NovAtel-like log based on data from a TIMEA log that contains
-- the UTC Month
    -- Inputs:
    --   TimeData_ String containing input data

    -- Returns:
    --   Custom Log String
--*************************************************************************
local function CreateMonthLog(TimeData_,OutputPort_)
  local HeaderData = TimeData_['Header']

  local MonthTable = {
'January','February','March','April','May','June','July','August','September','
October','November','December' }
  -- Setup the Header and Data.
  -- Leave out the leading # and trailing * as they are not included in the CRC
  local CustomLog =
    string.format("MONTHA,%s,%s,%s,%s,%s,%s,%s,%s,%s;%s",
```

```
                  OutputPort_, -- Note that the port is updated to the port where
this log will be sent
                  HeaderData['Sequence'],
                  HeaderData['IdleTime'],
                  HeaderData['TimeStatus'],
                  HeaderData['Week'],
                  HeaderData['Second'],
                  HeaderData['ReceiverStatus'],
                  HeaderData['Reserved'],
                  HeaderData['ReceiverSWVersion'],
                  MonthTable[tonumber(TimeData['UTCMonth'])])

  -- the crc32.lua script is included with the NovAtel Lua Dev Kit
  local CRC = require("crc32").CalculateBlock(CustomLog,0)

  -- Format together the leading #, the log data, the trailing * and calculated
CRC.
  return string.format("#%s*%08x",CustomLog,CRC)
end
--******************************************************************************
-- Request TIMEA logs on SCOM1, parse them and produce a new NovAtel-like custom
log
-- Inputs:
--   arg[1]     String representing the output port (e.g. 'COM1')
--******************************************************************************
local function main()

  local OutputPort = arg[1]

  if OutputPort == nil then
    print("No Output Port Specified")
    return
  end

  local SocketLib = require("socket")
  local SocketSCOM1 = SocketLib.udp()
  -- Setup the sockets
  local TargetIP = "127.0.0.1"

  assert(SocketSCOM1:setsockname("*",0))
  assert(SocketSCOM1:setpeername(TargetIP,require("scom").GetSCOMPort(1)))
  assert(SocketSCOM1:settimeout(3))

  -- Request the TIMEA log on SCOM1
  SocketSCOM1:send("LOG TIMEA ONTIME 1\r")
while true do
    -- Wait for TIMEA Logs
    local Buffer = SocketSCOM1:receive()
    if Buffer == nil then
      print("... timed out")
      break
    end

    local TimeData = ParseTIMEA(Buffer)

    if TimeData ~= nil then
```

```
      -- Uncomment the lines below to dump out the parsed TIMEA data
--      for Key,Value in pairs(TimeData) do
--        if type(Value) == "table" then
--          print(string.format("%s:",Key))
--          for SubKey,SubValue in pairs(Value) do
--            print(string.format(" %s: \"%s\"",SubKey,SubValue))
--          end
--        else
--          print(string.format("%s: \"%s\"",Key,Value))
--        end
--      end
--      print("------------------------\n")

      -- Format the new log
      local MonthLog = CreateMonthLog(TimeData,OutputPort)

      -- Send the log out the port
      -- Note in firmware version OM7MR0500RN0000 the SEND command can only
      -- send 100 bytes at once. That is sufficient for this example, but
      -- in an actual use case the log should be sent out in 100 byte chunks.
      SocketSCOM1:send(string.format('send %s \"%s\"\r',OutputPort,MonthLog))
    end
  end
end
--********************************************************************************
main()
```

3.3 Modules

Lua code can be located in multiple files and loaded as modules using the **`require`** function. Modules allow the user to group functionally related code in one file, and have other files import and use this functionality.

A module is loaded by passing in the name of the file without the .lua extension to the require function. A description of the **`require`** function can be found here: www.lua.org/pil/8.1.html.

The following example shows code from two files, mymodule.lua and use_mymodule.lua. These two script files can be packaged together and loaded onto the receiver using the steps in *Loading and running the application* on page 18.

```
-- File mymodule.lua
-- This is an example of creating a module called mymodule, which provides a
single function, mymodule.example_func()
-- which can be used by other scripts that import this module.

-- Create an empty table, which acts as the container for the module.
local mymodule = {}

-- Create a function that is available for the module.
function mymodule.example_func()
  print("Hello from mymodule.example_func()")
end

return mymodule

-- File use_mymodule.lua
-- Import the functionality from the file mymodule.lua.
local mymodule = require("mymodule")
```

```
print("Hello from use_mymodule.lua")
mymodule.example_func()
```

Files can also be placed in subdirectories and loaded by specifying the path to the file in the **`require`** function. The path is specified as the directory name followed by a . and appending the filename of the module without the .lua extension. The following example shows a module located in a subdirectory called testdir being loaded using the **`require`** function.

```
-- File /testdir/mymodule.lua

-- This is an example of creating a module called mymodule2, which provides a
single function, mymodule.example_func()
-- which can be used by other scripts that import this module.

local mymodule2 = {}

function mymodule2.example_func()
  print("Hello from mymodule2.example_func()")
end

return mymodule2
-- File use_mymodule2.lua
-- Import the functionality from the file mymodule2.lua. Note that the require
function
-- needs the testdir path to import the file correctly.

local mymodule2 = require("testdir.mymodule2")

print("Hello from use_mymodule2.lua")
mymodule2.example_func()
```

Additional information on Lua modules can be found here: lua-users.org/wiki/ModulesTutorial.

Chapter 4 Loading and running the application

Lua scripts can be deployed onto a NovAtel receiver and run using the **LUA** command (see page 32). The scripts are assembled into an ISO image, which is then written to a Data Block of the non-volatile storage within the receiver.

4.1 Packaging the application

In order to load Lua scripts onto a NovAtel receiver, the scripts must first be packaged up into a .hex file. Follow the steps below to create this package:

1. Place all the scripts to be loaded into a folder on a PC. This example will use C:\MYLUAPROJECT.
2. Download the NovAtel API.
3. Open a command prompt within the **utilities** directory of the Lua Dev Kit and use the **make_iso_hex.bat** batch file to create the .hex image. Usage for the script can be found by calling it with no arguments as shown below:

```
C:\luadevkit\utilities>make_iso_hex.bat
There are less than 4 arguments.
Usage: make_iso_hex.bat <source directory> <destination file> <version> <data
block> [platforms]
where:
     <source directory> - directory to be made into ISO file
     <destination file> - output path and filename
     <version> - version string for the output file, up to 15 characters
     <data block> - Flash DataBlock number, 0-7
     [platforms] - Optional list of supported platforms, separated by spaces.
                   Eg, OEM729 OEM7700 OEM7600
```

Here are some more details on the arguments:

Argument	Notes
<source directory>	This is the directory containing the Lua scripts
<destination file>	Full path name for the output file
<version>	User-determined version string to use for the .hex file. This string will be reported within the **VERSION** log the receiver
<data block>	Set this to 1
[platforms]	This is optional and can typically be left blank

Example:

```
C:\luadevkit\utilities>make_iso_hex.bat c:\myluaproject ..\debuglooplua.hex 1.00
1
Create ISO file...
Warning: creating filesystem that does not conform to ISO-9660.
Total translation table size: 0
Total rockridge attributes bytes: 0
Total directory bytes: 114
Path table size(bytes): 10
25 extents written (0 MB)

Create HEX file...

Set DataBlk...
*
* datablk - NovAtel Inc. data block utility n
* Executable Version: 2.28
* Header Version: 2
*
Processing \luadevkit\debuglooplua.iso.nodb.hex to \luadevkit\debuglooplua.hex
Success \luadevkit\debuglooplua.hex is ready to be programmed into flash.
```

ISO image limitations

There are a few limitations with the ISO image format used to package up the Lua scripts.

- There is a maximum directory depth of 8, including the root
- The maximum file name length is 27 characters plus a 4 character extension for a total of 31 characters
- The maximum directory name is 31 characters

4.2 Loading the application

Once the Lua scripts have been packaged up into a .hex file, they can be loaded onto the receiver. Use NovAtel Application Suite or SoftLoad commands to load the .hex file. Refer to Updating the firmware using NovAtel Application Suite or Updating Using SoftLoad Commands in the OEM7 Installation and Operation User Manual.

The presence of the Lua scripts can be verified as follows:

1. Check the **VERSION** log:

```
log version
<OK
[COM1]<VERSION COM1 0 90.5 UNKNOWN 0 138.554 02444020 3681 14581
<   3
<       GPSCARD "DDNRNNCBNA" "BMHR16370009M" "OEM7700-1.00"
"OM7MR0400AN0001" "OM7BR0000RB0000" "2018/Jan/09" "07:58:45"
<       OEM7FPGA "" "" "" "OMV070001RN0000" "" "" ""
<       DB_LUA_SCRIPTS "SCRIPTS" "Block1" "" "1.00" "" "2018/Jan/10"
"10:53:48"
[COM1]
```

If a Lua Scripts package has been loaded on to the receiver, it will be reported with a Component Type of DB_LUA_SCRIPTS. The "sw version" field reports the version string that was passed in to make_iso_hex.bat.

2. Check the **LUAFILESYSTEMSTATUS** log (see page 36).

```
log LUAFILESYSTEMSTATUS
```

```
<OK
[COM1]<LUAFILESYSTEMSTATUS COM1 0 89.5 UNKNOWN 0 0.194 02444020 b8f8
14581
<     MOUNTED ""
[COM1]
```

If the LUAFILESYSTEMSTATUS log reports that the file system is mounted, the ISO image within the package was successfully mounted. This happens automatically at system startup; there are no commands required to mount this file system.

3. Check the **LUAFILELIST** log (see page 35).

```
log LUAFILELIST
```

```
<OK
[COM1]<LUAFILELIST COM1 0 89.5 UNKNOWN 0 992.000 02444020 b447 14581
<     155 20180110 92730 "/lua/debugloop.lua"
[COM1]
```

If the LUAFILELIST log shows a file, it is available to the Lua interpreter.

4.3 Running the application

The **LUA** command (see page 32) is used to start the Lua interpreter.

To run a Lua script in the background, with no access to stdin, stdout and stderr, use **LUA START**.

To run the Lua interpreter in interactive mode with stdin, stdout and stderr connected to a receiver port, use **LUA PROMPT**.

The interpreter is started within the /lua working directory so scripts within that directory can be referenced directly, without a path.

4.3.1 Lua start

To execute a Lua script in the background use the LUA START option.

```
lua start helloworld.lua
```

```
<OK
[COM1]
```

```
log luastatus
```

```
<OK
[COM1]<LUASTATUS COM1 0 88.0 UNKNOWN 0 52.479 02444020 afcc 32768
<     0 "helloworld.lua" COMPLETED
[COM1]
```

```
log luaoutput
```

```
<OK
[COM1]<LUAOUTPUT 0 47.462
<     1 0 STDOUT "Hello World!"
<LUAOUTPUT 0 48.464
<    2 0 STDOUT "Hello again 1"
```

```
<LUAOUTPUT 0 49.465
<     3 0 STDOUT "Hello again 2"
<LUAOUTPUT 0 50.467
<     4 0 STDOUT "Hello again 3"
<LUAOUTPUT 0 51.468
<     5 0 STDOUT "Hello again 4"
<LUAOUTPUT 0 52.469
<     6 0 STDOUT "Hello again 5"
<LUAOUTPUT 0 52.470
<     7 0 STDOUT "Good Bye"
[COM1]
```

Note that the print statements within the script are output in the **LUAOUTPUT** log (see page 37). Also, note that the **LUASTATUS** log (see page 38) shows that the script has completed.

4.3.2 Lua prompt

To execute a Lua script with stdin, stdout and stderr connected to a receiver port, use the LUA PROMPT option. The print strings are output on the port where the **LUA** command (see page 32) was entered.

Example:

```
lua prompt helloworld.lua
Lua 5.3.4 Copyright (C) 1994-2017 Lua.org, PUC-Rio
Hello World!
Hello again 1
Hello again 2
Hello again 3
Hello again 4
Hello again 5
Good Bye
>
<OK
[COM1]

log luastatus

<OK
[COM1]<LUASTATUS COM1 0 88.0 UNKNOWN 0 52.479 02444020 afcc 32768
<     0 "helloworld.lua" COMPLETED
[COM1]

log luaoutput

<OK
[COM1]<LUAOUTPUT 0 47.462
<     1 0 STDOUT "Hello World!"
<LUAOUTPUT 0 48.464
<    2 0 STDOUT "Hello again 1"
<LUAOUTPUT 0 49.465
<     3 0 STDOUT "Hello again 2"
<LUAOUTPUT 0 50.467
<     4 0 STDOUT "Hello again 3"
<LUAOUTPUT 0 51.468
<     5 0 STDOUT "Hello again 4"
<LUAOUTPUT 0 52.469
<     6 0 STDOUT "Hello again 5"
<LUAOUTPUT 0 52.470
<     7 0 STDOUT "Good Bye"
[COM1]
```

On a different port (e.g. COM2) it can be seen that the INTERFACEMODE of COM1 has been changed to LUA.

```
log interfacemode
```

```
<OK
[COM2]<INTERFACEMODE COM2 29 97.0 UNKNOWN 0 25.700 0244c009 7a68 14581
<     COM1 LUA LUA OFF
...
```

The **LUASTATUS** log (see page 38) also shows that the script is executing.

```
log luastatus
```

```
<OK
[COM2]<LUASTATUS COM2 0 96.5 UNKNOWN 0 25.705 0244c009 afcc 14581
<     0 "-i helloworld.lua" EXECUTING
[COM2]
```

4.3.3 Single line Lua program

The "-e" option can be used to run a single line Lua program. Here is an example using a simple print call.

```
[COM1]lua prompt "-e print('Hello World')"
```

```
<OK
[COM1]Lua 5.3.4 Copyright (C) 1994-2017 Lua.org, PUC-Rio
Hello World
>
```

4.3.4 Passing arguments into Lua

To pass arguments into Lua, the script name and arguments must be enclosed within double quotes. The arguments are stored within the arg variable in Lua, which is a table of strings.

The example below shows how to iterate through the arguments and demonstrates some of the implications of the fact that the arguments are strings.

```
-- Print the script name
print(string.format('Script Name: "%s"',arg[0]))

FormatString = '%-10s%-10s%-15s%-15s'
print(string.format(FormatString,'Arg#','Type','String','Number'))

-- Iterate through the arguments
Sum = 0
NumberOfTwenties = 0
for i = 1,4 do
  -- Print some information about the argument
  -- NOTE: The type of these arguments is always "string"
  print(string.format(FormatString,i,type(arg[i]),arg[i],tonumber(arg[i])))

  -- Check if the string represents a number
  if (tonumber(arg[i]) ~= nil) then
     -- If the string represents a number, Lua will automatically
     -- convert the string to a number for arithmetic
     Sum = Sum + arg[i]
  end

  -- Since the arg values are always of type "string"
```

```
  -- a direct comparison with a number will always fail
  if (arg[i] == 20) then
     NumberOfTwenties = NumberOfTwenties + 1
  end
end
print('')
print(string.format("Sum of Number Arguments: %d",Sum))
print(string.format("Number of 20s found: %d",NumberOfTwenties))
```

Here is how to call this script using the **LUA** command (see page 32). Note how the string "20" is not considered equal to the number 20.

```
lua prompt "scriptargs.lua 1 20 Hello 300"

<OK
[COM1]Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
Script Name: "scriptargs.lua"
Arg#      Type      String         Number
1         string    1              1
2         string    20             20
3         string    Hello          nil
4         string    300            300
Sum of Number Arguments: 321
Number of 20s found: 0
>
```

4.3.5 Starting a script automatically

To start Lua automatically when the receiver boots, add a file named "autoexec.lua" to the root directory of the lua script package. This script will be executed when the receiver starts up. To run other scripts from the autoexec.lua script, use the **dofile** Lua command as shown in the example below.

Here is the content of an example hello.lua script:

```
Person1 = arg[1]
Person2 = arg[2]

print(string.format("%s says hello to %s",Person1,Person2))
```

Here is the content of an example autoexec.lua script:

```
arg[1] = "Alice"
arg[2] = "Bob"

dofile("hello.lua")
```

The autoexec.lua script sets up the command line arguments for the hello.lua script and then runs the script.

Here is the example in action:

```
log luastatus onchanged

<OK
[COM1]<LUASTATUS COM1 0 87.5 UNKNOWN 0 0.614 02444020 afcc 32768
<     0 "autoexec.lua" COMPLETED
[COM1]

log luaoutput onchanged

<OK
```

```
[COM1]<LUAOUTPUT 0 0.593
<     1 0 STDOUT "Alice says hello to Bob"
[COM1]saveconfig

<OK
[COM1]

reset

<OK
[COM1]
[COM1]<LUASTATUS COM1 0 13.0 UNKNOWN 0 1.234 02440000 afcc 32768
<     0 "autoexec.lua" COMPLETED
[COM1]<LUAOUTPUT 0 1.151
<     1 0 STDOUT "Alice says hello to Bob"
[COM1]

log luafilelist

<OK
[COM1]<LUAFILELIST COM1 1 84.5 UNKNOWN 0 32.000 02444020 b447 32768
<     55 20180613 105553 "/lua/autoexec.lua"
<LUAFILELIST COM1 0 87.5 UNKNOWN 0 32.000 02444020 b447 32768
<     97 20180613 105502 "/lua/hello.lua"
[COM1]
```

Chapter 5 Debugging and testing

5.1 ZeroBrane Studio

ZeroBrane Studio is a lightweight Integrated Development Environment (IDE) for Lua. A version of ZeroBrane is included in the NovAtel API that contains additions to make creating and debugging Lua scripts on NovAtel receivers easier.

The ZeroBrane Studio project website is studio.zerobrane.com/.

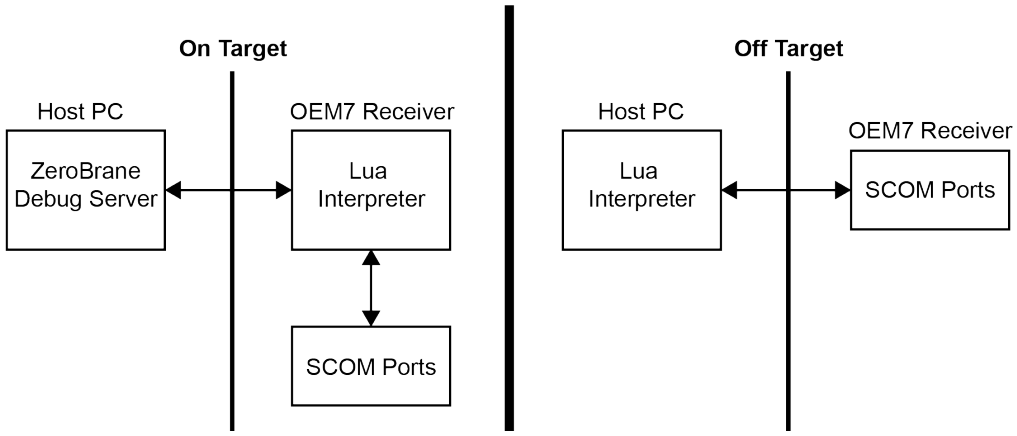
The ZeroBrane documentation on debugging can be found at: studio.zerobrane.com/doc-remote-debugging.

5.2 On target vs. off target debugging

There are two main ways to debug Lua scripts for use on the OEM7 receiver:

- On Target:
  The Lua interpreter on NovAtel receivers can be debugged using the ZeroBrane IDE via an Ethernet connection to the receiver. In this method, the Lua interpreter on the target (i.e. the receiver) is executing the script and the Lua interpreter on the host PC is just providing a debug server.
- Off Target:
  The Lua interpreter within ZeroBrane Studio can execute a script and interact with the receiver via the SCOM ports over an Ethernet connection.

The diagram below describes how the various pieces interact in both methods and the table that follows contains more notes on the differences:



Debugging type	Lua interpreter	IP address to use for SCOM	Notes
On Target	On OEM7 Receiver	127.0.0.1	This environment more closely resembles how the Lua scripts will be deployed in an end user use case.
Off Target	On PC	IP Address of OEM Receiver See the **IPSTATUS** log	Useful for quickly developing Lua scripts and testing non-real time aspects of the code.

SCOM port numbers

The NovAtel provided `scom` module can be used to programatically determine socket port numbers for the SCOM port. For more information, see *Additions and limitations* on page 30.

See the table below for the port numbers:

SCOM port	Port number
SCOM1	49154
SCOM2	49155
SCOM3	49156
SCOM4	49157

5.3 On target debugging

Version 1.70 of the ZeroBrane IDE is included within the NovAtel API under the zerobrane folder. This version has been customized for use with NovAtel receivers. However, the stock version can be downloaded here: studio.zerobrane.com/support.

5.3.1 Prerequisites

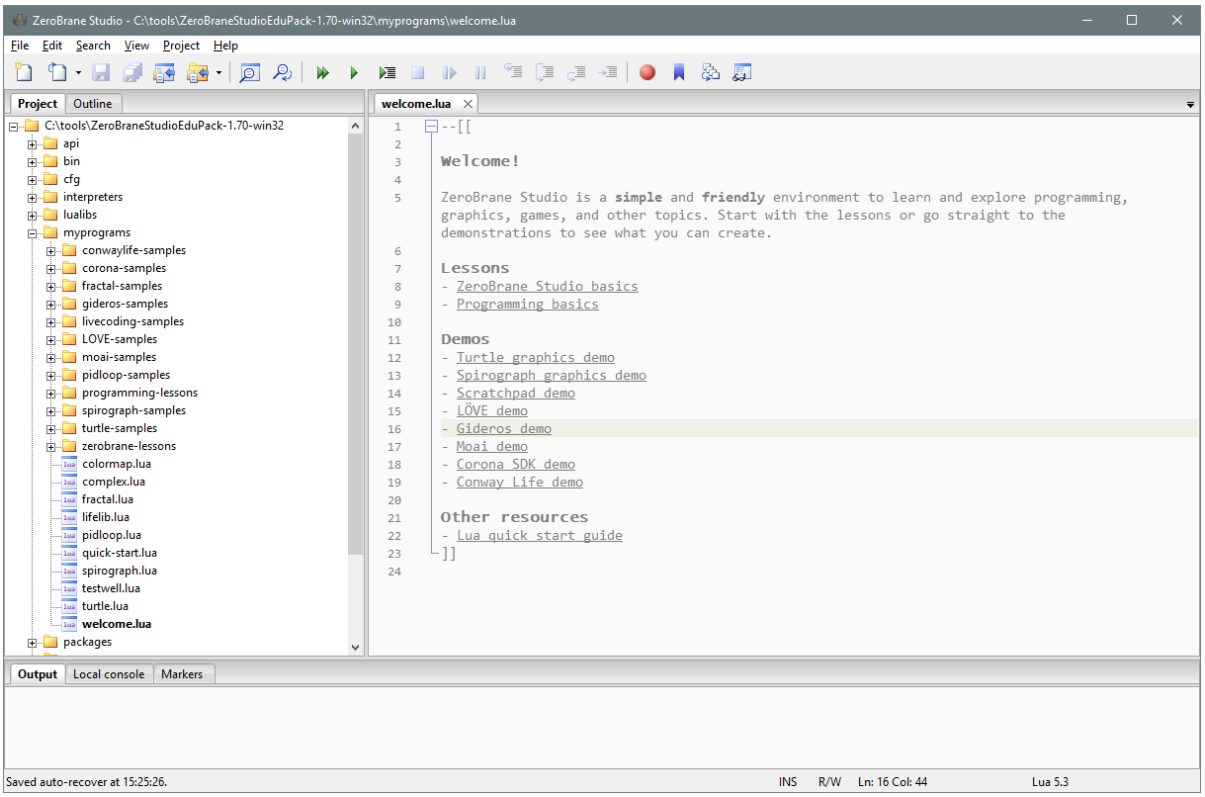
Here are the prerequisites to enable NovAtel receiver on-target Lua debugging:

- An OEM7 receiver running firmware 7.05.00 or later.
- A firmware model supporting the user application.
- An Ethernet connection from a host PC to the receiver.
- The script to debug must be available on both the PC and the OEM7 receiver.

Note that the `mobdebug.lua` script, which is used by ZeroBrane for remote debugging, is preloaded into the Lua interpreter and does not need to be added to the Lua script package that is loaded onto the receiver.

5.3.2 PC and receiver setup

1. Download and install the ZeroBrane IDE to a PC.
2. Run ZeroBrane IDE (zbstudio.exe). The IDE will open up to a default Project containing some examples.



3. Switch to the Lua 5.3 - NovAtel interpreter, which is the version running on the OEM7. To do this, select **Project** | **Lua Interpreter** | **Lua 5.3 - NovAtel**.
4. The Local console tab at the bottom can be used to experiment with the Lua syntax.

   Example:

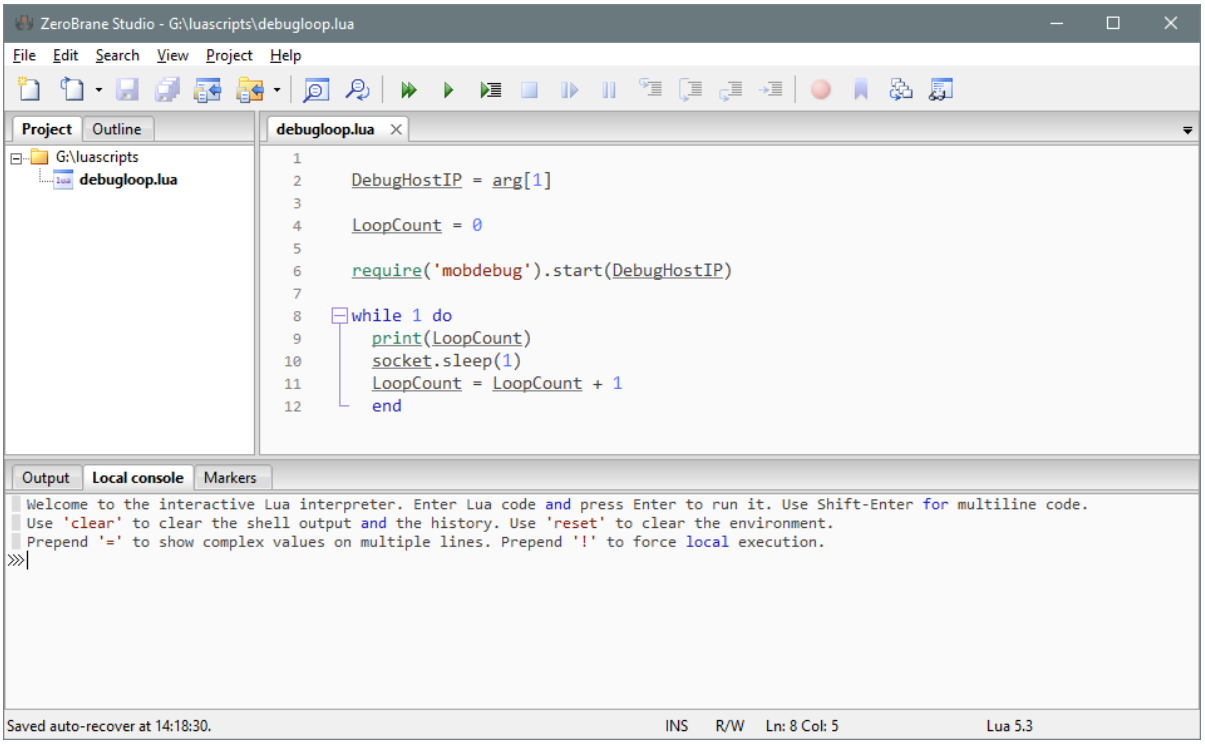
   ```
   Welcome to the interactive Lua interpreter. Enter Lua code and press
   Enter to run it. Use Shift-Enter for multiline code.
   Use 'clear' to clear the shell output and the history. Use 'reset' to
   clear the environment.
   Prepend '=' to show complex values on multiple lines. Prepend '!' to
   force local execution.
   MyVar = 123
   MyOtherVar = 456
   print(MyVar+MyOtherVar)
   579
   ```

5. On the OEM7 receiver, configure the network using the **ETHCONFIG** command and **IPCONFIG** command, and verify that a connection is possible. For example, use a terminal program to connect to an ICOM port and request a **VERSION** log. For details on how to set this up see Ethernet communications in the OEM7 Installation and Operation User Manual.
6. On the PC where the ZeroBrane IDE is running, set the location where the scripts that were loaded onto the receiver can be found. To do this select **Project** | **Project Directory** | **Choose** and select the folder. The contents of the selected folder will be packaged and loaded onto the receiver, later in step 9. This example uses the simple **`debugloop.lua`** script shown here:\

   ```
   DebugHostIP = arg[1]

   LoopCount = 0
   ```

```
require('mobdebug').start(DebugHostIP)

while 1 do
  print(LoopCount)
  socket.sleep(1)
  LoopCount = LoopCount + 1
  end
```

7. Place a break-point in the script by clicking to the left of the code line as shown below:



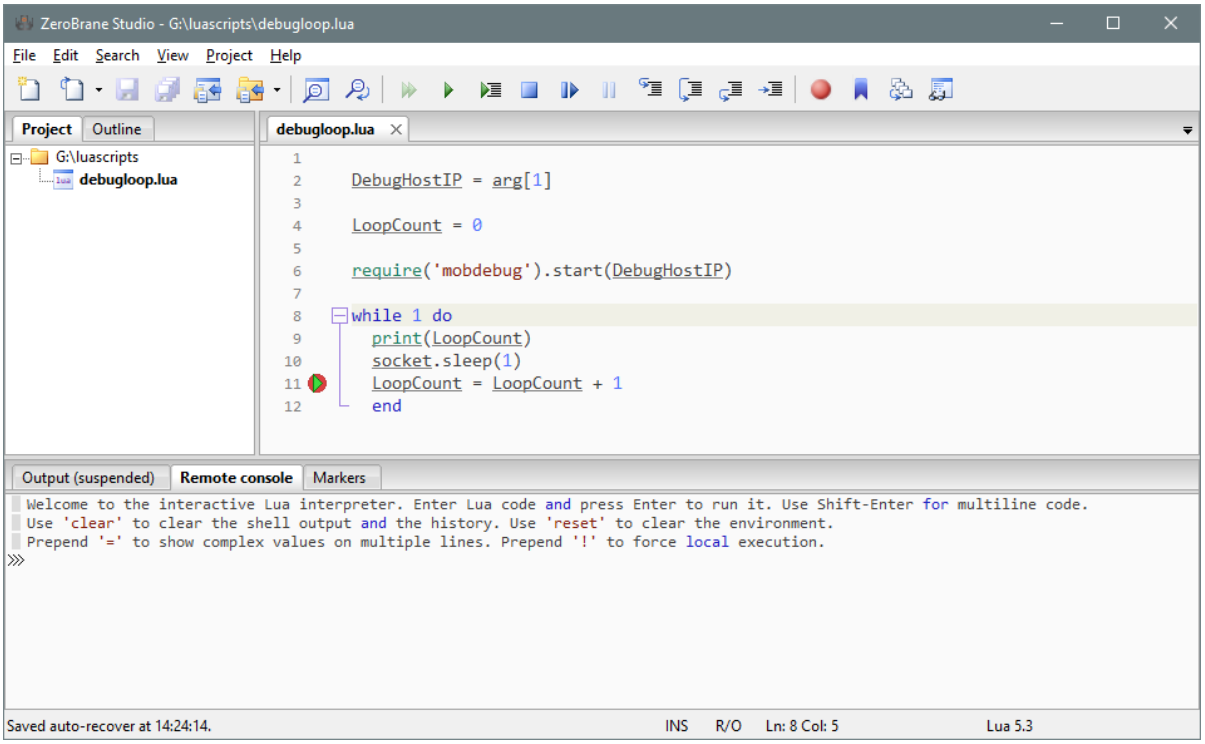
8. Turn on the ZeroBrane debug server. To do this ensure that **Project** | **Start Debug Server** is checked.
9. Create a Lua script package from the project directory and load it onto the receiver using the steps described in *Loading and running the application* on page 18.
10. The OEM7 receiver should now have the same Lua script available to it as the ZeroBrane IDE does. Verify this using the **LUAFILESYSTEMSTATUS** log (see page 36) and **LUAFILELIST** log (see page 35).

```
log luafilesystemstatus onchanged

<OK
[COM1]<LUAFILESYSTEMSTATUS COM1 0 91.0 UNKNOWN 0 0.343 02444020 b8f8
14581
<     MOUNTED ""

log luafilelist

<OK
[COM1]<LUAFILELIST COM1 0 89.0 UNKNOWN 0 54.000 02444020 b447 14581
<     176 20180122 141649 "/lua/debugloop.lua"
[COM1]
```

11. Start the Lua script, passing the IP address of the PC running ZeroBrane, as the argument. The Lua interpreter will reach out to the debugger running on the host PC to establish the debugging connection.

```
lua prompt "debugloop.lua 198.161.68.53"
<OK
[COM1]Lua 5.3.4  Copyright (C) 1994-2017Lua.org, PUC-Rio
0
1
2
3
4
5
```

The ZeroBrane IDE will then be able to control the Lua interpreter on the receiver:



12. Use the debugging controls within the IDE to step through the code and set break points.

Chapter 6 Additions and limitations

This chapter describes some of the ways that the Lua interpreter running on the OEM7 receiver is different than a standard Lua interpreter.

6.1 Additions

- The mobdebug module is preloaded to facilitate debugging.
- Help messages are available for some functions. Use the **`H()`** function within the receiver Lua prompt to view help.
- The crc32 module was created by NovAtel to generate CRCs for NovAtel messages. Use **`H(crc32)`** on the receiver Lua prompt for more details.
- The scom module was created by NovAtel as convenience functions to access the SCOM ports. Use **`H(scom)`** on the receiver Lua prompt for more details.
- If a Lua script has been started with the **LUA PROMPT** command, it can be stopped using the **`os.exit()`** Lua command.
- A 64 kB RAM disk has been provided to the Lua interpreter at the location /tmp.
- The **`os.tmpname()`** function will return a unique file name within /tmp.
- Some environment variables have been added to the Lua interpreter running on the OEM7 receiver. They can be accessed using the **`os.getenv()`** function and are defined as follows:
  - "ONTARGET" is set to "true".
  - "GPSCARD_PSN" is set to the receiver PSN.
  - "ENCLOSURE_PSN" is set to the receiver's enclosure PSN, if one is set.

6.2 Limitations

- The C User Application is not available to customers. That is, customers cannot write C code or take external, compiled libraries and link them to the Lua interpreter running on the OEM7 receiver.
- The Operating System Library is not fully-functioning.
  - **`os.time()`** and **`os.date()`** report in Greenwich Mean Time (GMT)
  - **`os.date()`** will report time starting from Jan 1 1970, until GPS coarse time is set, at which point it will report the current time.
- There is no way to stop a Lua script that was started with **LUA START**, unless the script itself completes.
- It is not recommended to use LUA for high rate logging. Expect latency of 50 ms or more.
- Latency will increase when idle time is low.

Chapter 7 Lua commands

The following commands are used with Lua.

- **LUA** command on the next page

7.1 LUA

Configure Lua interpreter

Platform: **OEM719, OEM729, OEM7500, OEM7600, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7**

Use this command to configure the execution of the Lua interpreter on the receiver. Scripts that appear within the **LUAFILELIST** log (see page 35) can be executed by the Lua interpreter.

Message ID: 2181

Abbreviated ASCII syntax:

```
LUA option [LuaInterpreterArguments]
```

Abbreviated ASCII example:

```
lua start "printarguments.lua 1 2 3 4 5"
```

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
1	Command header	-	-	LUA header	-	H	0
2	option	START	1	Start the Lua interpreter in the background. The file descriptors stdout, stdin and stderr will not be accessible outside the receiver.	Enum	4	H
		PROMPT	2	Start the Lua interpreter in interactive mode and connect stdout, stdio and stderr to the port on which the command was entered. The INTERFACEMODE of that port will be changed to LUA for both RX and TX.			
3	LuaInterpreter Arguments	STRING		String containing Lua interpreter options including the name of the script file to run and arguments to pass to the script. This string must be enclosed in quotes if it contains any spaces. String arguments within the field **must** be enclosed by single quotes.	String [400]	Variable	H+4

The format of the Lua Interpreter Arguments is as follows as adapted from the standard Lua 5.3 interpreter:

```
[options] [script [args]]
Available options are:
  -e stat  execute string 'stat'
  -i       enter interactive mode after executing 'script'.
           (This is added to the arguments when using the PROMPT option of the
           LUA command)
  -l name  require library 'name'
```

Chapter 8 Lua logs

The following logs are used with Lua.

- **LUAFILELIST** log on the next page
- **LUAFILESYSTEMSTATUS** log on page 36
- **LUAOUTPUT** log on page 37
- **LUASTATUS** log on page 38

8.1 LUAFILELIST

List available Lua scripts

Platform: **OEM719, OEM729, OEM7500, OEM7600, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7**

This sequenced log informs the user of the available scripts, obtained from the ISO loaded onto the receiver. The size of the file, last change date in yyyymmdd format, last change time in hhmmss format, and path to the files are printed as well.

Message ID: 2151

Log type: Polled

Recommended input:

```
LOG LUAFILELIST
```

Abbreviated ASCII example:

```
[COM1]<LUAFILELIST COM1 6 89.5 UNKNOWN 0 4.000 02444020 b447 14635
<     0 20180202 151403 "/lua/uppercase.lua"
<LUAFILELIST COM1 5 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     2706 20180129 152042 "/lua/debugloop.lua"
<LUAFILELIST COM1 4 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     4692 20180202 110107 "/lua/parsetime.lua"
<LUAFILELIST COM1 3 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     4764 20180205 105415 "/lua/scom_rx.lua"
<LUAFILELIST COM1 2 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     3728 20180202 104830 "/lua/scomtunnel.lua"
<LUAFILELIST COM1 1 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     3044 20180201 144849 "/lua/scriptargs.lua"
<LUAFILELIST COM1 0 90.5 UNKNOWN 0 4.000 02444020 b447 14635
<     2337 20180129 155140 "/lua/sendtocom2.lua"
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	LUAFILELIST header	-	H	0
2	Size	File size (in Bytes)	Ulong	4	H
3	Date	Last change date When viewed as a string, the date is of the form YYYYMMDD. So, numerically, the date is (Year * 10000) + (Month * 100) + (Day).	Ulong	4	H+4
4	Time	Last change time When viewed as a string, the time is HHMMSS. So, numerically, the time is (Hour * 10000) + (Minute * 100) + (Second).	Ulong	4	H+8
5	Path	The path to the Lua script The maximum length of this string is 256 bytes.	String	Variable	H+12

8.2 LUAFILESYSTEMSTATUS

Query mount status of Lua scripts

Platform: **OEM719, OEM729, OEM7500, OEM7600, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7**

Use this log to query the mount status of the ISO image that contains the Lua scripts loaded on to the receiver.

Message ID: 2150

Log type: Asynch

Recommended input:

```
LOG LUAFILESYSTEMSTATUS
```

Abbreviated ASCII example:

```
<LUAFILESYSTEMSTATUS COM1 0 90.0 UNKNOWN 0 0.204 02444020 b8f8 14635
<     MOUNTED ""
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	LUAFILESYSTEMSTATUS header		H	0
2	Status	The status of the file system. See *Table 1: File system status* below.	Enum	4	H
3	Error	String that indicates the error message if mounting fails The maximum length of this string is 52 bytes.	String	Variable	H+4

Table 1: File system status

Value	Description
1	UNMOUNTED
2	MOUNTED
3	BUSY
4	ERROR
5	UNMOUNTING
6	MOUNTING

8.3 LUAOUTPUT

Output stderr and stdout from the Lua interpreter

Platform: **OEM719, OEM729, OEM7500, OEM7600, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7**

Use this log to output **`stderr`** and **`stdout`** messages from the Lua interpreter.

The **LUAOUTPUT** log uses a short header.

Message ID: 2240

Log type: Asynch

Recommended input:

```
LOG LUAOUTPUT ONNEW
```

Abbreviated ASCII example:

```
<LUAOUTPUT 0 346044.929
<     1 0 STDOUT "Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio"
<LUAOUTPUT 0 346044.987
<     2 0 STDOUT "> "
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	LUAOUTPUT header	-	H	0
2	Sequence Number	Running number of each LUAOUTPUT log produced by the system	Ulong	4	H
3	Executor Number	Lua Executor Number that produced the data	Ulong	4	H+4
4	Data Source	See *Table 2: Lua data source* below	Enum	4	H+8
5	Data	NULL-terminated string containing a single line of data from stderr or stdout. This string is not terminated with a carriage return or line feed. This string contains only printable characters. The maximum length of this string is 128 bytes.	String	Variable	H+12

Table 2: Lua data source

Binary	ASCII	Description
0	STDOUT	Data is from stdout
1	STDERR	Data is from stderr

8.4 LUASTATUS

Display status of Lua scripts

Platform: **OEM719, OEM729, OEM7500, OEM7600, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7**

Use this log to determine which scripts are running on the receiver and whether the scripts have exited or encountered errors.

Message ID: 2181

Log type: Collection

Recommended input:

```
LOG LUASTATUS
```

Abbreviated ASCII example:

```
[COM1]<LUASTATUS COM1 1 84.5 FINESTEERING 1963 402110.866 02400000 2e18 32768
<     0 "icom_rx.lua 127.0.0.1 3001" EXECUTING
<LUASTATUS COM1 0 84.5 FINESTEERING 1963 402110.866 02400000 2e18 32768
<     1 "" NOT_STARTED
```

The example above is for the projected log output for two executors.

Field	Field type	Description	Format	Binary bytes	Binary format
1	Log header	LUASTATUS header		H	0
2	Number	Executor number	Ulong	4	H
3	Script	Script and arguments	String [256]	Variable	H+4
4	Status	Script status. See *Table 3: Script status* below.	Enum	4	Variable

Table 3: Script status

Binary	ASCII	Description
0	NOT_STARTED	There is no script running on the executor
1	EXECUTING	The script is running
2	COMPLETED	The script completed successfully
3	SCRIPT_ERROR	The script exited with an error
4	EXECUTOR_ERROR	The script executor encountered an error while attempting to run the script

Chapter 9 Using Lua to access I/O devices connected to the receiver

A common request from users of OEM7 receivers is have the receiver interact with other equipment in the embedded system. For example, manipulating GPIOs to control external devices or monitor other sensors. To meet this need, OEM7 receivers provide access to the:

- I2C bus
- CAN bus

9.1 I2C bus

OEM7600, OEM7700 and OEM7720 receivers have I2C bus signals available which allows connections to a variety of devices.

“The I2C bus was designed by Philips in the early ’80s to allow easy communication between components which reside on the same circuit board. Philips Semiconductors migrated to NXP in 2006”. (i2c-bus.org)

OEM7 receivers that provide access to the I2C signals support three commands and one log to interact with I2C devices connected to the receiver:

- **USERI2CBITRATE** command (see page 41)
- **USERI2CREAD** command (see page 42)
- **USERI2CWRITE** command (see page 44)
- **USERI2CRESPONSE** log (see page 46)

In the *SampleScripts* folder of the NovAtel API you will find two examples of I2C “drivers” for GPIO expanders. One example is for the MCP23008 8 I/O port expander (mcp23008ioe.lua) and the other example is for the PCA9554 8-bit I/O expander (PCA9554ioe.lua). These examples can be used as the basis for creating drivers to interact with other I2C devices.

9.2 CAN bus

All OEM7 receivers support connection to the CAN bus. The receiver provides access through either the upper level application protocol (J1939 and NMEA2000) or the user CAN access feature. For information about using the upper level application protocol, see CAN bus communications in the OEM7 Documentation Portal (docs.novatel.com/OEM7).

The receiver does support access to the upper level application protocol and user CAN access at the same time. If the upper level application protocol has been enabled using the **CANCONFIG** command, the user CAN access commands will be rejected.

The user CAN access feature supports three commands and two logs.

- **USERCANCLOSE** command (see page 48)
- **USERCANOPEN** command (see page 49)
- **USERCANWRITE** command (see page 51)

- **USERCANDATA** log (see page 53)
- **USERCANSTATUS** log (see page 55)

9.3 USERI2CBITRATE

Set the communication rate for the User I2C bus

Platform: OEM7600, OEM7700, OEM7720

Use this command to set the communication bit rate for the User I2C bus.

Message ID: 2383

Abbreviated ASCII syntax:

```
USERI2CBITRATE BitRate
```

Factory default:

```
USERI2CBITRATE 400K
```

ASCII examples:

```
USERI2CBITRATE 100K
USERI2CBITRATE 400K
```

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
1	Command header	–	–	USERI2CBITRATE header	–	H	0
2	BitRate	100K	1	Set the User I2C bus communication rate to 100 kbits/s.	Enum	4	H
		400K	2	Set the User I2C bus communication rate to 400 kbits/s.			

9.4 USERI2CREAD

Read data from devices on the I2C bus

Platform: OEM7600, OEM7700, OEM7720

Use this command to read data from devices on the I2C bus.

This command only applies to OEM7 receivers that have I2C signals available on the interface connector. The compatible receivers are listed in the **Platform** section above.

The **USERI2CRESPONSE** log (see page 46) can be used to check the completion or status of the read operation. An optional user defined Transaction ID can be provided to help synchronize requests with responses in the **USERI2CRESPONSE** log (see page 46). This command is primarily intended to be used by Lua applications that need to interact with external devices.

Reading from an I2C device requires a device address, to distinguish which physical device is to be accessed, a register within the device, and the expected number of bytes to be read. Depending on the type of I2C device, register addresses can be 1 to 4 bytes in length, so the actual number of bytes for the register address must be specified.

For some I2C devices there are no registers within the device. In this case, the Register Address Length is 0 and no bytes are supplied for the Register Address.

The **USERI2CREAD** command is flexible to handle all of these situations.

Message ID: 2232

Abbreviated ASCII syntax:

```
USERI2CREAD DeviceAddress RegisterAddressLen RegisterAddress RequestReadLen
[TransactionID]
```

Examples:

```
USERI2CREAD 70 1 AB 12 1234
USERI2CREAD 74 3 ABCDEF 234 5678
USERI2CREAD 74 0 234 5678
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Command header	USERI2CREAD header	-	H	0
2	DeviceAddress	The 7 bit address of the I2C device. Valid values are 0 through 127. For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits. There is no 0x prefix and spaces are not allowed in the string.	Uchar	1[1]	H

1. In the binary case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.

Field	Field type	Description	Format	Binary bytes	Binary offset
3	RegisterAddressLen	The length of the register address that follows. Valid values are 0 through 4.	Ulong	4	H+4
4	RegisterAddress	The actual address of the register to be read. The number of bytes here must match the RegisterAddressLen. In particular, when RegisterAddressLen is 0, this field is empty (even for a binary command) For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits for each byte in the register address. There is no 0x prefix and spaces are not allowed in the string.	Uchar Array	X[1]	H+8
5	RequestReadLen	The length of data expected to be retrieved from the device. Valid values are 1 through 256.	Ulong	4	H+12[1]
6	TransactionID	An optional user provided ID for this transaction. Default = 0. This transaction ID will be copied to the **USERI2CRESPONSE** log (see page 46) created for this read operation.	Ulong	4	H+16[2]

1. H+8 if X=0
2. H+12 if X=0

9.5 USERI2CWRITE

Write data to device on I2C bus

Platform: OEM7600, OEM7700, OEM7720

Use this command to write data to devices on the I2C bus.

This command only applies to OEM7 receivers that have I2C signals available on the interface connector. The compatible receivers are listed in the **Platform** section above.

The **USERI2CRESPONSE** log (see page 46) can be used to check the completion or status of the write operation. An optional user defined Transaction ID can be provided to help synchronize requests with responses in the **USERI2CRESPONSE** log (see page 46). This command is primarily intended to be used by Lua applications that need to interact with external devices.

Writing to an I2C device requires a device address, to distinguish which physical device is to be accessed, a register within the device and the data. Depending on the type of I2C device, register addresses can be 1 to 4 bytes in length, and so the actual number of bytes for the register address must be specified.

For some I2C devices there are no registers within the device. In this case, the Register Address Length is 0, and no bytes are supplied for the Register Address.

For some other I2C devices, write operations are done in two stages:

1. The first stage sends a write command with a register address, but no data. This is a dummy write to set the register within the device for write operations that follow.
2. The second stage sends a write command with no register address, but does send a stream of data.

The **USERI2CWRITE** command is flexible to handle all of these situations.

Message ID: 2233

Abbreviated ASCII syntax:

```
USERI2CWRITE DeviceAddress RegisterAddressLen RegisterAddress WriteDataLength
WriteData [TransactionID]
```

Examples:

```
USERI2CWRITE 70 1 AB 12 313233343536373839 3A3B3C 1234
USERI2CWRITE 74 3 ABCDED 5 1234567890 1234
USERI2CWRITE 40 0 5 1234567890 1234
USERI2CWRITE 40 2 AABB 0 1234
```
(a dummy write)

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Command header	USERI2CWRITE header	-	H	0

Field	Field type	Description	Format	Binary bytes	Binary offset
2	DeviceAddress	The 7 bit address of the I2C device. Valid values 0 through 127. For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits. There is no 0x prefix and spaces are not allowed in the string.	Uchar	1[1]	H
3	RegisterAddressLen	The length of the register address that follows. Valid values are 0 through 4.	Ulong	4	H+4
4	RegisterAddress	The actual address of the register to be written. The number of bytes here must match the RegisterAddressLen. In particular, when RegisterAddressLen is 0, this field is empty (even for a binary command) For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits for each byte in the register address. There is no 0x prefix and spaces are not allowed in the string.	Uchar Array	X[1]	H+8
5	WriteDataLength	The length of data to be written in bytes. Valid values are 0 through 256.	Ulong	4	H+12[2]
6	WriteData	The data to be written. The number of bytes in this data block must match the WriteDataLength. In particular, when WriteDataLength is 0, this field is empty. For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits for each byte in the data block. There is no 0x prefix and spaces are not allowed in the string. Data is streamed to the device as a series of bytes in the order provided.	Uchar Array	Y[3]	H+16[4]
7	TransactionID	An optional user provided ID for this transaction. Default = 0. This transaction ID will be copied to the **USERI2CRESPONSE** log (see page 46) created for this write operation.	Ulong	4	H+16+4*INT ((Y+3)/4)[5]

1. In the binary case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.
2. H+8 if X=0
3. In the binary case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.
4. H+12 if X=0
5. H+12+4*INT((Y+3)/4) if X=0

9.6 USERI2CRESPONSE

Status of USERI2CREAD or USERI2CWRITE Command

Platform: OEM7600, OEM7700, OEM7720

This log reports the status of a previously executed **USERI2CREAD** or **USERI2CWRITE** command. There is one log emitted for each command that is executed.

For the **USERI2CREAD** command (see page 42), this log outputs the data read from the device on the I2C bus and the status of the read operation.

For the **USERI2CWRITE** command (see page 44), the status of the write operation is reported and the data field will always be 0.

Message ID: 2234

Recommended input:

```
log USERI2CRESPONSE onnew
```

Abbreviated ASCII example 1:

```
USERI2CREAD 70 4 aabbccdd 12 6789

<USERI2CRESPONSE COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     70 aabbccdd OK READ 6789 12 000102030405060708090a0b
```

Abbreviated ASCII example 2:

```
USERI2CWRITE 70 3 aabbcc 8 0001020304050607 12345

<USERI2CRESPONSE COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     70 aabbcc OK WRITE 12345 0
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	USERI2CRESPONSE header	-	H	0
2	DeviceAddress	The 7 bit address of the I2C device. Valid values are 0 through 127. For ASCII and Abbreviated commands, this field is a hexadecimal string of two digits. There is no 0x prefix and spaces are not allowed in the string.	Uchar	1[1]	H
3	RegisterAddress	The actual register address used for the operation. This is a ULONG value in hexadecimal format (without 0x prefix).	Ulong	4	H+4
4	ErrorCode	Error code for the operation. See *Table 4: Error code* on the next page.	Enum	4	H+8
5	OperationMode	Operation mode code. See *Table 5: Operation mode code* on the next page.	Enum	4	H+12

1. In the binary case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.

Field	Field type	Description	Format	Binary bytes	Binary offset
6	TransactionID	This is the copy of Transaction ID provided to the command.	Ulong	4	H+16
7	ReadDataLength	For a Read operation, this is the actual number of bytes read from the I2C device. For a Write operation, this value is always zero.	Ulong	4	H+20
8	ReadData	For a Read operation, this is the data read from the device. For ASCII logs this field is displayed as a string of hexadecimal digits, with two digits per byte. The first byte retrieved from the I2C device is the first byte displayed and so on. The maximum size of this field is 256 bytes. When ReadDataLength is zero, this field will be empty.	HEXBYTE ARRAY	Y	H+24

Table 4: Error code

Binary	ASCII	Description
0	OK	I2C transaction is successful
1	IN_PROGRESS	I2C transaction is currently in progress
2	DATA_TRUNCATION	I2C transaction read data was truncated
3	BUS_BUSY	I2C bus is busy
4	NO_DEVICE_REPLY	No device replied to the I2C transaction request
5	BUS_ERROR	I2C bus error or bus arbitration lost
6	TIMEOUT	I2C transaction has timed out
7	UNKNOWN_FAILURE	I2C transaction has an unexplained failure

Table 5: Operation mode code

Binary	ASCII	Description
0	NONE	No Operation
1	READ	Read Operation
2	WRITE	Write Operation
3	SHUTDOWN	Shut down Operation

9.7 USERCANCLOSE

Disable a CAN port

Platform: OEM719, OEM729, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7

Use this command to disable a CAN port. The **USERCANSTATUS** log (see page 55) can be used to check the completion or status of the operation. An optional user defined Transaction ID can be provided to help synchronize requests with responses in the **USERCANSTATUS** log (see page 55).

This command is primarily intended to be used by Lua applications that need to interact with external devices.

Closing a CAN port requires the port ID. There are up to 2 CAN buses available on OEM7 receivers, CAN1 and CAN2.

Message ID:2313

Abbreviated ASCII syntax:

```
USERCANCLOSE PortID [TransactionID]
```

Abbreviated ASCII examples:

```
USERCANCLOSE CAN1 1234
USERCANCLOSE CAN2
```

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
1	Command header	–	–	USERCANCLOSE header	–	H	0
2	PortID	CAN1	1	The CAN port Identifier.	Enum	4	H
		CAN2	2				
3	TransactionID			An optional user provided ID for this transaction. Default = 0. This transaction ID will copied to the **USERCANSTATUS** log (see page 55) created for this operation.	Ulong	4	H+4

9.8 USERCANOPEN

Enable a CAN port

Platform: OEM719, OEM729, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7

Use this command to enable a CAN port. The **USERCANSTATUS** log (see page 55) can be used to check the completion or status of the operation. An optional user defined Transaction ID can be provided to help synchronize requests with responses in the **USERCANSTATUS** log (see page 55).

This command is primarily intended to be used by Lua applications that need to interact with external devices.

Opening a CAN port requires the port ID. There are up to 2 CAN buses available on OEM7 receivers, CAN1 and CAN2. The speed (i.e. bit rate) must also be specified.

For the CAN hardware to select certain messages on the bus, a set of hardware filters need to be associated with the port.

This command is rejected if the specified CAN port has already been opened. The CAN port can be opened by either a previous **USERCANOPEN** command or a **CANCONFIG** command.

Message ID: 2312

Abbreviated ASCII syntax:

```
USERCANOPEN PortID BitRate ElementNum [HWFilters] [TransactionID]
```

Abbreviated ASCII examples:

```
USERCANOPEN CAN1 250K 3 EXT 11111111 00000000 EXT 12222222 00000000 STD 333
00000000 123456
```

```
USERCANOPEN CAN2 100K 0 123457
```

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
1	Command header	–	–	USERCANOPEN header	–	H	0
2	PortID	CAN1	1	The CAN port Identifier.	Enum	4	H
		CAN2	2				
3	BitRate	See *Table 6: CAN bit rate* on the next page		The CAN speed.	Enum	4	H+4
4	ElementNum	0 - 16		Number of HWFilter Arrays	Ulong	4	H+8

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset

5	HWFilters			An array of class CANHwFilter with the following structure:
				`{`
				`CanMessageTypeEnum eMessageType;`
				`ULONG ulAcceptCode;`
				`ULONG ulAcceptMask;`
				`}`
				Only filters in use need to be provided, so the format for this section (for N elements) is:
				{eMessageType1 ulAcceptCode1 ulAcceptMask1 ... eMessageTypeN ulAcceptCodeN ulAcceptMaskN}
				If N = 0, this whole field is omitted.
				N is the number of elements in the array, which is specified in the *ElementNum* field.
					HWFilter Array	12*N	H+12

6	TransactionID			An optional user provided ID for this transaction. Default = 0. This transaction ID will copied to the **USERCANSTATUS** log (see page 55) created for this operation.	Ulong	4	H+12+ 12*N

Table 6: CAN bit rate

Binary	ASCII	Description
0	10K	10 Kbits/sec
1	20K	20 Kbits/sec
2	50K	50 Kbits/sec
3	100K	100 Kbits/sec
4	125K	125 Kbits/sec
5	250K	250 Kbits/sec
6	500K	500 Kbits/sec
7	1M	1 Mbits/sec

9.9 USERCANWRITE

Write frames to a CAN port

Platform: OEM719, OEM729, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7

Use this command to write a basic CAN frame to the specified CAN port. The **USERCANSTATUS** log (see page 55) can be used to check the completion or status of the operation. An optional user defined Transaction ID can be provided to help synchronize requests with responses in the **USERCANSTATUS** log (see page 55).

This command is primarily intended to be used by Lua applications that need to interact with external devices.

Writing a CAN frame requires a Port ID, which indicates the CAN bus to write to; a identifier type (STANDARD or EXTENDED) and an identifier number. The operation can be set as "blocked" (TimeOutMS > 0) or "non-blocked" (TimeOutMS == 0). If set to "blocked", the operation will wait a certain period of time for a slot in the TX FIFO buffer. For system health reasons, there is no option for the operation to be blocked indefinitely.

Message ID: 2255

Abbreviated ASCII syntax:

```
USERCANWRITE PortID MessageType MessageID Datalength Data [TimeOutMS]
[TransactionID]
```

Abbreviated ASCII examples:

```
USERCANWRITE CAN1 STD 123 8 3132333435363738 12 1234
USERCANWRITE CAN2 EXT 1767 3 ABCDED
```

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
1	Command header	–	–	USERCANWRITE header	–	Header	0
2	PortID	CAN1	1	The CAN port Identifier.	Enum	4	H
		CAN2	2				
3	MessageType	STD	1	Standard message	Enum	4	H+4
		EXT	2	Extended message			
4	MessageID			The Standard message ID has 11 bits. The Extended message ID has 29 bits.	HexUlong	4	H+8
5	Datalength	0 to 8		The length of data to be written in bytes.	Ulong	4	H+12

Field	Field type	ASCII value	Binary value	Description	Format	Binary bytes	Binary offset
6	Data			The data to be written. The number of bytes in this data block must match the *DataLength*. In particular, when *DataLength* is 0, this field is empty. For ASCII and Abbreviated ASCII commands, this field is a hexadecimal string of two digits for each byte in the data block. There is no 0x prefix and spaces are not allowed in the string. Data is streamed to the device as a series of bytes in the order provided.	UChar Array	N[1]	H+16
7	TimeOutMS	0 to 5000		The number of milliseconds for the operation to timeout. Default is 0. If set to 0, there is no wait for a free slot in the TX FIFO. If no slot is available, the data drops and a flag is set in **USERCANSTATUS** log (see page 55). Maximum value is 5000.	Ulong	4	H+16+4*INT ((N+3)/4)
8	TransactionID			An optional user provided ID for this transaction. Default = 0. This transaction ID is copied to the **USERCANSTATUS** log (see page 55) created for this write operation.	Ulong	4	H+20+4*INT ((N+3)/4)

1. In the binary command/log case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.

9.10 USERCANDATA

Basic frame data from a CAN port

Platform: OEM719, OEM729, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7

The **USERCANDATA** log reports the received CAN basic frame data. Depending on the number of frames in the RX FIFO buffer, there will be multiple **USERCANDATA** logs published. There is an index field indicating the order of the frames received.

USERCANDATA messages are asynchronously published based on Message ID matches set by the acceptance filters of the **USERCANOPEN** command (see page 49).

Message ID: 2316

Recommended input:

```
log USERCANDATA onnew
```

Abbreviated ASCII examples:

```
<USERCANDATA COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     CAN1 STD 178 8 AABBCCDDEEFF1122 123456 0
<USERCANDATA COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     CAN1 STD 178 5 3344556677 123456 1
<USERCANDATA COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     CAN1 EXT 1F567 8 1122334455667788 123456 2
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	USERCANDATA header	–	H	0
2	Port ID	CAN port ID. See *Table 7: CAN port ID* on the next page.	Enum	4	H
3	Message Type	Message type. See *Table 8: CAN message type* on the next page.	Enum	4	H+4
4	Message ID	The Standard message ID has 11 bits. The Extended message ID has 29 bits.	HexUlong	4	H+8
5	DataLength	The length of data to be written in bytes. Valid range is 0 to 8 inclusive.	Ulong	4	H+12
6	Data	The data of this frame.	HexByte Array	N[1]	H+16
7	TransactionID	The Transaction ID provided to the command.	Ulong	4	H+16+4*INT ((N+3)/4)

1. In the binary command/log case, additional bytes of padding are added after this field to maintain 4-byte alignment for the fields that follow.

Field	Field type	Description	Format	Binary bytes	Binary offset
8	Frame Index	The frame order.	Ulong	4	H+20+4*INT ((N+3)/4)

Table 7: CAN port ID

Binary	ASCII	Description
1	CAN1	CAN port 1
2	CAN2	CAN port 2

Table 8: CAN message type

Binary	ASCII	Description
1	STD	Standard message
2	EXT	Extended message

9.11 USERCANSTATUS

Status of commands sent to a CAN port

Platform: OEM719, OEM729, OEM7700, OEM7720, PwrPak7, CPT7, CPT7700, SMART7

The **USERCANSTATUS** log reports the status of a previously executed **USERCANOPEN**, **USERCANCLOSE** or **USERCANWRITE** command. There is one log output for each command that is executed.

For the **USERCANWRITE** command (see page 51), the *Number of Frames* field of the **USERCANSTATUS** log indicates the number of frames placed into the TX FIFO buffer, 0 or 1.

Message ID: 2315

Recommended input:

```
log USERCANSTATUS onnew
```

Abbreviated ASCII examples:

```
USERCANWRITE CAN1 STANDARD 0x234 8 ABCDEF0123456789 23 234578
<USERCANSTATUS COM1 0 84.0 FINESTEERING 1994 257885.895 02000000 e3f6 32768
<     CAN1 WRITE 234578 OK 1
```

Field	Field type	Description	Format	Binary bytes	Binary offset
1	Log header	USERCANSTATUS header	–	H	0
2	Port ID	The CAN port Identifier. See *Table 9: CAN port ID* below.	Enum	4	H
3	Operation	Operation performed. See *Table 10: Operation mode* on the next page.	Enum	4	H+4
4	TransactionID	This is the copy of Transaction ID provided to the command.	HexUlong	4	H+8
5	Status	Status message. See *Table 11: CAN status message* on the next page.	Enum	4	H+12
6	Number of Frames	For a WRITE operation, this is the number of frames placed in the TX FIFO (0 - 1). For a READ operation, this is the number of frames available in RX FIFO (0 - 32).	Ulong	4	H+16

Table 9: CAN port ID

Binary	ASCII	Description
1	CAN1	CAN port 1
2	CAN2	CAN port 2

Table 10: Operation mode

Binary	ASCII	Description
0	NONE	No operation
1	OPEN	Open operation
2	CLOSE	Close operation
3	READ	Read operation
4	WRITE	Write operation

Table 11: CAN status message

Binary	ASCII	Description
0	OK	CAN operation is successful.
1	FAIL	CAN operation failed.
2	PortAlreadOpened	Attempt to open a CAN port that is already open
3	PortAlreadyClosed	Attempt to close a CAN port that is already closed
4	TX FIFO full	Unable to write a frame to the TX FIFO as the buffer is full
5	RX FIFO empty	No frame available to read from the RX FIFO


HEXAGON
NovAtel