ICON-5066-1.0

# Icon-5066 Administration Guide

Isode

# Table of Contents

# 1 Software version

This guide is published in support of Isode Icon-5066 1.0. It may also be pertinent to later releases. Please consult the release notes for further details.

# 2 Readership

This guide is intended for administrators or integrators who plan to configure and manage the Isode Icon 5066 server and associated tools.

# 3 How to use this guide

All evaluators and, in particular, systems integrators are advised to read the following chapters before attempting to configure an Icon-5066 system:

- Chapter 1, *Overview* Overview of the features, components and architecture.
- Chapter 2, *Installing the Software* How to install the software.
- Chapter 3, *Icon-5066 Configuration* How to configure the system. Not all of this will be required reading, for example it is only necessary to understand the configuration corresponding to modems in use.

# 4 Typographical conventions

The text of this manual uses different typefaces to identify different types of objects, such as file names and input to the system. The typeface conventions are shown in the table below.

| Object | Example |
|---|---|
| File and directory names | *isoentities* |
| Program and macro names | **mkpasswd** |
| Input to the system | `cd newdir` |
| Additional information to note, or a warning that the system could be damaged by certain actions. | Notes are additional information; cautions are warnings. |

Arrows are used to indicate options from the menu system that should be selected in sequence.

For example, **File** → **New** means to select the **File** menu and then select the **New** option from it.

# 5 Filesystem placeholders

A number of directory names are given in the text, and the actual locations (below) vary depending on the target platform.

| Name | Place holder for the directory used to store... | Windows (default) | UNIX |
|---|---|---|---|
| *(ETCDIR)* | System-specific configuration files. | *C:\Isode\etc* | */etc/isode* |
| *(SHAREDIR)* | Configuration files that may be shared between systems. | *C:\Program Files\Isode\share* | */opt/isode/share* |
| *(BINDIR)* | Programs run by users. | *C:\Program Files\Isode\bin* | */opt/isode/bin* |
| *(SBINDIR)* | Programs run by the system administrators. | *C:\Program Files\Isode\bin* | */opt/isode/sbin* |
| *(EXECDIR)* | Programs run by other programs; for example, M-Switch channel programs. | *C:\Program Files\Isode\bin* | */opt/isode/libexec* |
| *(LIBDIR)* | Libraries. | *C:\Program Files\Isode\bin* | */opt/isode/lib* |
| *(DATADIR)* | Storing local data. | *C:\Isode* | */var/isode* |
| *(LOGDIR)* | Log files. | *C:\Isode\log* | */var/isode/log* |

# 6 Support queries and bug reporting

A number of email addresses are available for contacting Isode. Please use the address relevant to the content of your message.

- For all account-related inquiries and issues: customer-service@isode.com. If customers are unsure of which list to use then they should send to this list. The list is monitored daily, and all messages will be responded to.
- For all licensing related issues: license@isode.com.
- For all technical inquiries and problem reports, including documentation issues from customers with support contracts: support@isode.com. Customers should include relevant contact details in initial calls to speed processing. Messages which are continuations of an existing call should include the call ID in the subject line. Customers without support contracts should not use this address.
- For all sales inquiries and similar communication: sales@isode.com.

Bug reports on software releases are welcomed. These may be sent by any means, but electronic mail to the support address listed above is preferred. Please send proposed fixes with the reports if possible. Any reports will be acknowledged, but further action is not guaranteed. Any changes resulting from bug reports may be included in future releases.

Isode sends release announcements and other information to the Isode News email list, which can be subscribed to from the address: http://www.isode.com/company/contact.html

# Chapter 1 Overview

This chapter introduces the Isode Icon 5066 server, giving an overview of its main features and components.

## 1.1 What is Icon-5066?

Icon-5066 is a high performance software implementation of the link level services of STANAG 5066 Edition 3 standard for digital data communication over HF radio. It is designed to be independent of Modem and ALE (Automatic Link Establishment) products, and can be extended to support new products as desired. The software suite consists of a number of components, including the STANAG 5066 protocol server, Web based configuration, logging and support and test tools. Icon-5066 can support any application that connects with the STANAG 5066 SIS protocol, including a number of other Isode products.

## 1.2 Target Audience

This document is largely aimed at people who are deploying, configuring or testing Icon-5066 services, though the Monitoring section is likely to be helpful for operators.

## 1.3 Software Overview

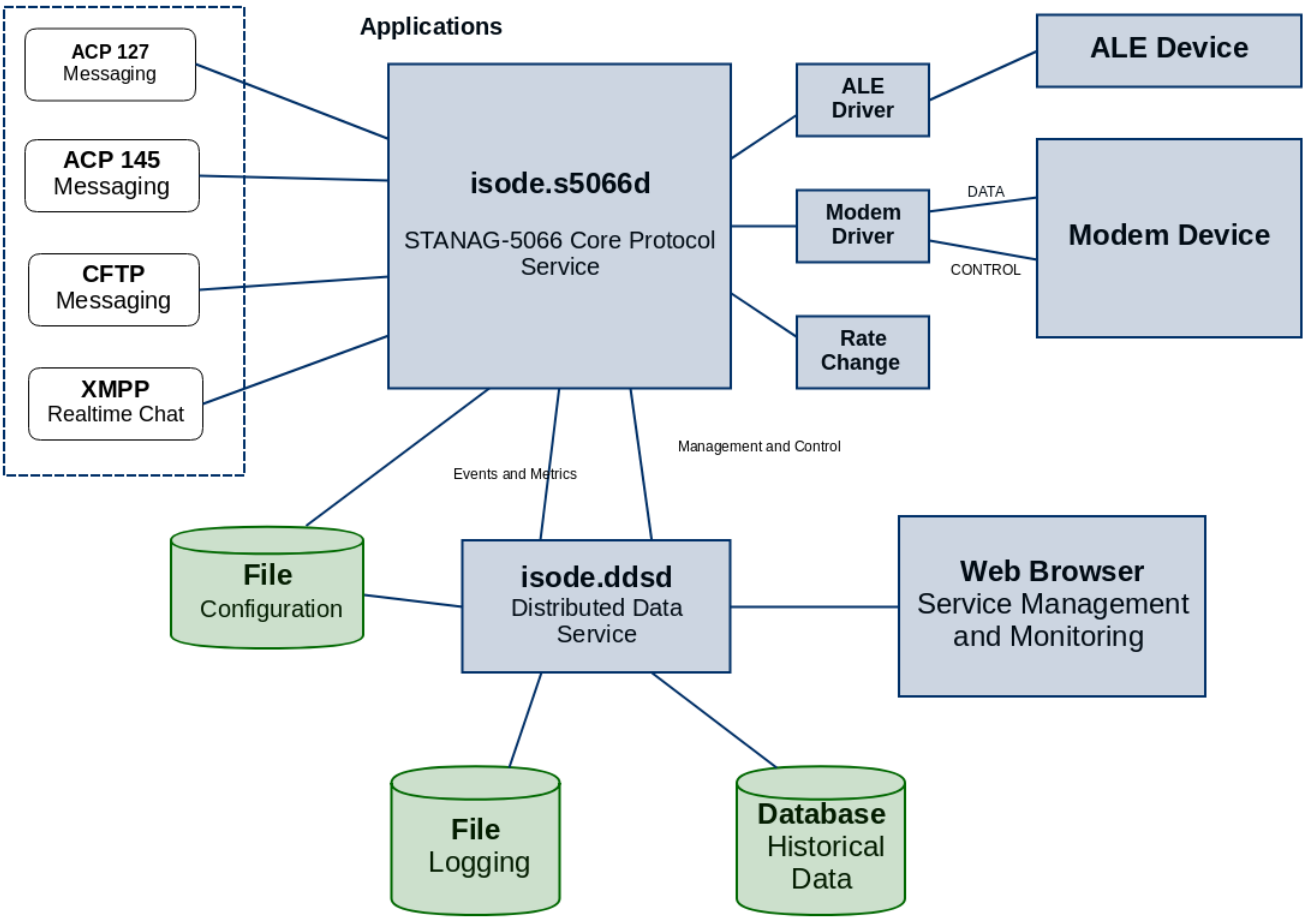
The product consists of two server components:

- **Distributed Data Service (isode.ddsd)** This server provides a data service that handles logging, monitoring and configuration.
- **Core Protocol Server (isode.s5066d)** This server provides the core service, which runs the S5066 nodes.

The software suite also includes two test tools:

- **MoRaSky** Modem/radio/sky combination simulator which can be used to test Icon-5066 in the absence of hardware modems and radios.
- **HF Tool** Tool for testing modem drivers independently of Icon-5066.

**Figure 1.1. Device control options**




This diagram provides an overview of the architecture, and how the various components interact.

# Chapter 2 Installing the Software

This chapter describes how to install the Icon-5066 software, and then gives a brief overview on how to manage the services provided.

## 2.1 Installing the Software

Detailed information on how to install the software is found in the Release Notes, which are available on the Isode Web site. An overview of the steps is:

1. Install prerequisite software. This includes:
   a. Java runtime (required by Morasky).
   b. Visual C++ Redistributable (Windows only).
2. Install the Icon-5066 software package.

### 2.1.1 Managing Windows Services

Once you have installed the Icon5066 software you need to create the Icon-5066 Distributed Data Service and Icon-5066 Core Protocol Server services. From the Start Menu -> All -> Programs -> Icon-5066 1.0 -> Services -> Install Icon-5066 Services. Note that some versions of Windows may require you to run this with the "Run as Administrator" option. Once the services (**Icon-5066 Core Protocol Server** and **Icon-5066 Distributed Data Service**) have been created you can manage them using the Windows **Services** application. You can also use the Web-based configuration application to stop and start the **Icon-5066 Core Protocol Server**.

### 2.1.2 Managing Linux Services

A Linux service for the Distributed Data Service is added to the system with the package installation and the service can be started as follows:

```
% systemctl start isode.icon.ddsd
```

Similar standard systemd commands can be used to stop, restart or query the status of the service. The Core Protocol Server must be configured before installing the corresponding service via the Web-based configuration UI.

### 2.1.3 Service Management with the Web Console

The Icon-5066 Distributed Data Service must be managed as a system process. The Icon-5066 Core Service can also be managed through the Web-based configuration and monitoring console. This is described in the next section.

# Chapter 3 Icon-5066 Configuration

This section covers the creation of an Icon-5066 configuration in detail, together with reference information on all configuration options.

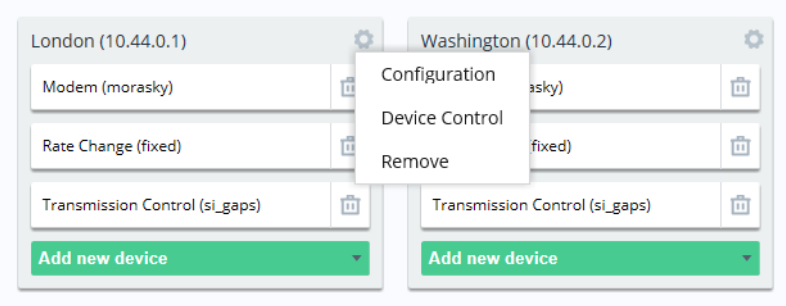
## 3.1 Management Console

The Icon-5066 Management Console is a Web interface used to configure and monitor an Icon-5066 service. The `isode.ddsd` (Distributed Data Service Daemon) service hosts the Icon-5066 Management Console Web user interface, so this should be started first. The console is served on port 4001, and it can be accessed from a Web browser with a URL as per the examples below:

```
http://myserver.hostname:4001
```

```
http://localhost:4001
```

**Figure 3.1. Node configuration panels**

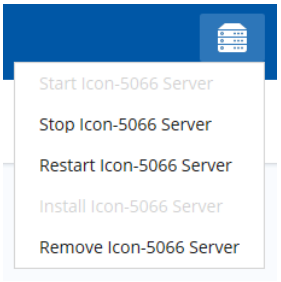



The working area of the main screen (example above) consists of a number of panels, with each representing a single S5066 node. Each node consists of a core configuration together with a number *devices* that are used by or control the behaviour of the S5066 node. Each working node configuration must define the following set of mandatory devices:

- **Modem** device that defines how to communicate with the underlying modem (simulated or real). See Section 3.7.2, “Modem Device Types and Configuration”.
- **Rate Change** device controls the rate at which data can be sent to the modem. See Section 3.7.3, “Rate Change Devices”.
- **Transmission Control** device controls the pattern of data transmission. See Section 3.7.4, “Transmission Control ”.

A working configuration must be defined before starting the `isode.s5066d` STANAG-5066 protocol server for the first time. Also note that the management console can be used to control the s5066d service as shown here:

**Figure 3.2. Service management menu**



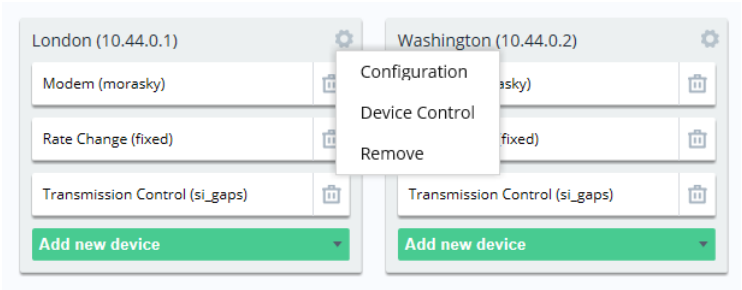

The Icon-5066 server loads the configuration for all nodes on start-up. After changes are made to a node, the Icon-5066 server must be restarted for the changes to take effect.

The next section takes the reader through the set of steps required to add a node, and to configure the set of devices associated with the node.

## 3.2 Adding a Node

An Icon-5066 Node is added by selecting the **Add New Node** button. This will open the Node Configuration dialog window, which is described in detail in the next section. Once a node has been created a new node information panel is shown. The node information panel is headed by a node description and the STANAG 5066 node address, together with the set of devices associated with the node. In the example below, two nodes have been configured where each node has modem, rate change and transmission control devices configured. To edit an existing node configuration, click on the cog icon in the top right of a node information panel and then select the **Configuration** item.

**Figure 3.3. Node configuration**




## 3.3 Deleting a Node

A node can be deleted by selecting the cog icon at the top right of the node panel and then choosing the **Remove** option.

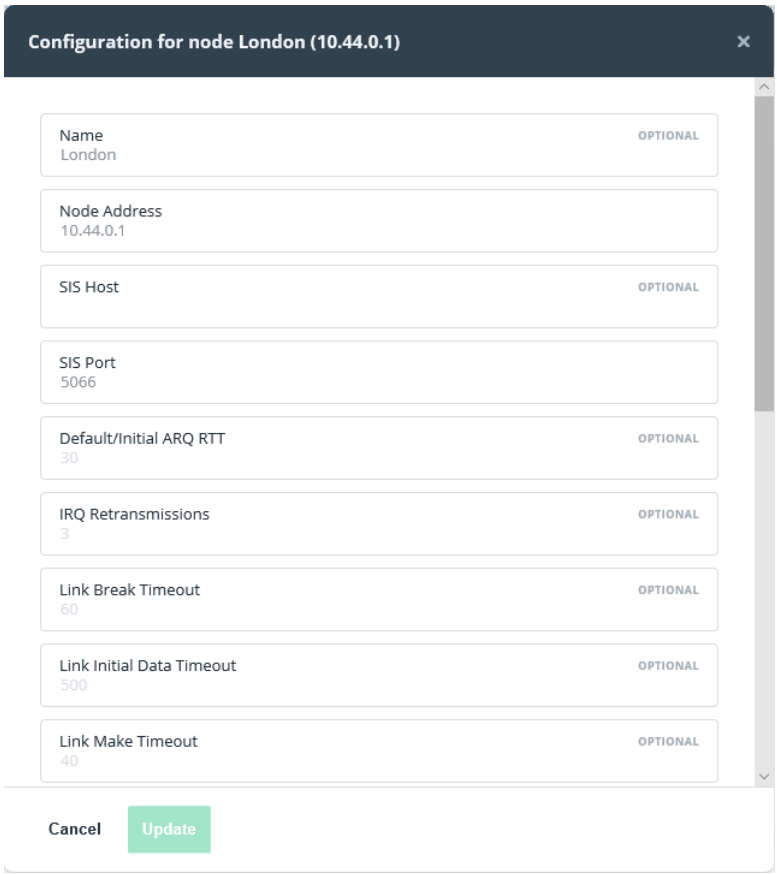
## 3.4 Activating Configuration Changes

The following sections describe how to modify the configuration of a node. Configuration changes are activated by a restart of the core Icon-5066 server. The UI marks clearly the number of pending changes. The server can be restarted by selecting the **Restart Icon-5066 Server** option from the server management menu displayed in the top right of the main screen.

## 3.5 Node Configuration Dialog

The Node Configuration dialog, shown below, is used to define a new node or to configure an existing one. This dialog is opened by selecting the **Configuration** option from the cog icon in the node panel.

**Figure 3.4. Node configuration**




The configuration options are as follows:

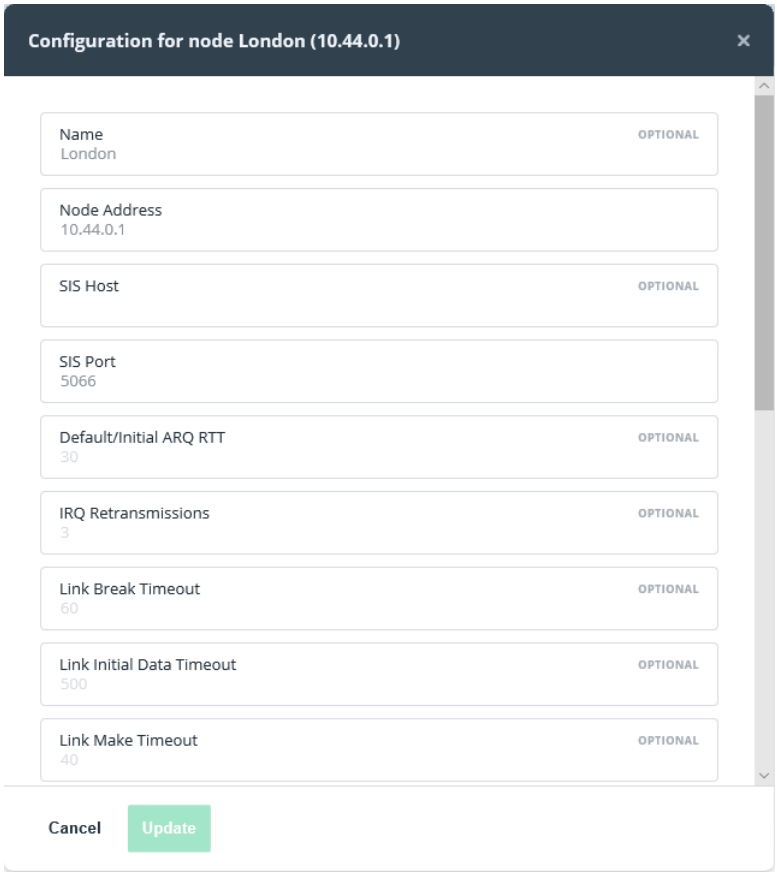
1. **Name** A description used to identify and label the node (e.g. "London").
2. **Node Address** The STANAG 5066 Address of the node, which is 3.5 bytes represented in a dotted quad notation. The maximum value is 31.255.255.255. This must be specified.

3. **SIS Host** This controls the network addresses on which the SIS service for this node listens. The default is to listen on all IPv4 and IPv6 addresses, and so will not usually need to be set.
4. **SIS Port** The port on which the SIS service listens and the value must be unique across all nodes running on one host. The default value is 5066, which is the standard value.
5. **Default/Initial ARQ RTT** For duplex operation only. How long an ACK may be outstanding before a D_PDU is resent in full duplex mode. Default is 30 seconds.
6. **IRQ Retransmissions** IRQ (Idle Repeat Request) is a robust handshaking mechanism used for window resync and for legacy (non-auto-baud) data rate change. This is the number of times to retransmit after a timeout. The default of 3 is recommended.
7. **Line Break Timeout** Timer used by the responder when setting up a CAS-1 soft link. Responder will initiate link break if no data received after this time.
8. **Link Initial Data Timeout** If at least one *data* D_PDU is not received on a newly activated physical link this timer specifies the period of time after which the called node shall abort the physical link and declare it inactive.
9. **Link Make Timeout** Timer used by CAS-1 soft link initiator. If no response received on link initiation after this timer the initiator will try again to make the link. Default is 30 seconds.
10. **Implicit Soft Links** If this is set, CAS-1 soft links are established implicitly. It is recommended to set this when ALE is used.
11. **Link Establishment Retries** Count used by CAS-1 initiator to control the number of attempts to make a link. Default is 3.
12. **Max non-Exclusive Links** The maximum number of CAS-1 links that can be active at any time. The default is 1, which is recommended where interoperability with other servers is required, as use of higher values is quite likely to lead to issues. For deployments only using Icon-5066, a value larger than the number of servers on the network is recommended.
13. **MAX TTL** The maximum TTL (Time To Live) allowed for a SIS APDU. This will override any larger values requested by a SIS client. Default is 1800 seconds (30 minutes).
14. **MTU Default** Default is 2048.This is the maximum size allowed. It is recommended to not modify this.
15. **Queue Low Water Mark** Threshold used to control SIS flow control on a per-destination basis. The default is 20480 bytes.
16. **Softlink Idle Timeout** CAS-1 soft links are timed out after this interval if no data is sent or received within this time interval. The default is 400 seconds. Where multiple CAS-1 soft links are configured, a longer value may be appropriate. Where only one soft link is allowed, it is recommended to reduce this value to 0.
17. **Win Resync Threshold** This setting is used to control the threshold of re-synchronisation of the selective-repeat ARQ protocol which operates on the link between the source and destination nodes. A drop PDU is sent for each expired PDU until this threshold is reached. The default is 10.
18. **Disable Type-14 DPDU Padding** If set then padding DPDUs (which are specified in S5066-EP2) are disabled and replaced with padding bytes. Setting this decreases reliability and degrades performance. This may need to be set for interoperability with servers that do not support S5066-EP2.
19. **U_PDU Confirmation maximum fragment size** The maximum size of user data contained in a delivery confirmation or rejection sent to a peer node. Default 100. A larger size increases overhead. A smaller size increases risk of ambiguity.
20. **Enable binary dump of RX/TX streams** Enable binary dump of RX/TX streams. If set then all data sent to and received from each modem is recorded. This data may be useful for subsequent analysis and perfomance. When setting this option, privacy and disk usage issues should be considered.

# 3.6 Device Control Options

The device control option specifies per-node configuration used to control how the core protocol server interacts with the devices at a higher level. Device control is configured by selection the cog and choosing **Device Control**. The configuration options provided are:

**Figure 3.5. Device control options**



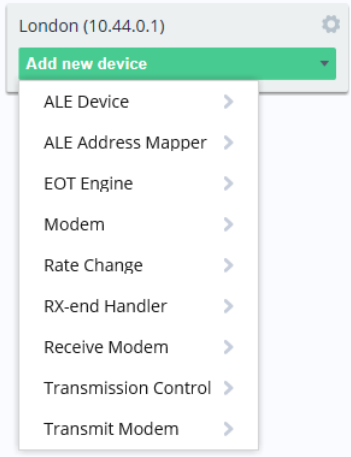

- **Duplex mode** Options are:
  - **half** Half-duplex (simplex) mode, using a single radio channel. This is the default.
  - **full** Full duplex mode, using two radio channels.
  - **bcast** Broadcast only mode.
  - **recv** Receive only and never transmit.
- **LFSN mode** Long Frame Sequence Number mode allows a much larger ARQ window, improving throughput for around 2400bps and above, and especially for wideband transmission rates. This follows the "STANAG-5066 Large Window Support" (S5066-EP5) extension. Default is off. It is recommended to set this is speeds will be 2400bps or higher and if this is supported by peers.
- **Verbosely log all LUA VM calls** To enable verbose logging of calls to the devices when debugging the system.
- **Byte in Hex to use for padding the start of a transmission** Specifies the byte that will be repeated if leading padding is used. Default is FF.
- **Minimum number of bytes of padding to insert at the start of the transmission** Controls padding length by number of bytes.

- **Byte sequence in hex to output after the padding and before the transmission** Default is nothing. RapidM RC66 uses “90”. Must be an even number of digits to represent an exact number of bytes.
- **Byte in hex to use for padding within the transmission** It may be necessary to use padding within a transmission in continuous mode. This specifies the byte to use. Default is 55.
- **Byte sequence in hex to output after the final transmission and before any padding transmission** Default is nothing. RapidM RC66 uses “FFFF”. Must be an even number of digits to represent an exact number of bytes.
- **Byte in hex to use for padding after the transmission** Specifies the byte that will be repeated if trailing padding is used. default FF.
- **Minimum number of bytes of padding to insert after the end of the transmission** Controls padding length by number of bytes.
- **Minimum number of milliseconds of padding to insert after the end of the transmission** Controls padding length by time.
- **EOW 4 Capability to send as 2 hex digits** STANAG 5066 Ed3 C.5.4 specifies a Capability Advertisement EOW (Engineering Order Wire) message. If a value is set here, this capability will be advertised to peer systems.
- **Number of seconds (or fractions) to delay the EOT expiry event** The EOT (End of Transmission) time for a transmission is determined by from the transmission. This parameter can be used to add a delay to when the node will consider starting a transmission. This may be needed where the sending node is slow to switch from transmit to receive.
- **Non-ARQ Rx-window minimum for simplex in seconds** Default 180. Non-ARQ transmissions are numbered and the window can wrap. At high speeds, this wrapping may lead to ambiguity for the receiver. This time specifies the minimum time before wrapping. Simplex transmission needs to allow for a transmission where the repeat of a D_PDU happens in a different transmission.
- **Non-ARQ Rx-window minimum for broadcase or duplex in seconds** Default 30. Non-ARQ transmissions are numbered and the window can wrap. At high speeds, this wrapping may lead to ambiguity for the receiver. This time specifies the minimum time before wrapping. For Broadcast and duplex transmission this should be chosen so that disruption of the whole period is unlikely.

## 3.7 Devices

Once a node is created, devices needed to be added to the node in order to provide desired functionality. The device selection menu is shown below.

**Figure 3.6. Add new device menu**




The currently supported device types are:

- **Modem** a single modem that is used by a node for data transmission and/or reception. The modem may be used in duplex or simplex mode.
- **Transmit Modem** a modem that is used solely for data transmission. A **Receive Modem** device will need to be configured if this node will also receive data.
- **Receive Modem** a modem that is used solely for data reception. A **Transmit Modem** device will need to be configured if this node will also transmit data.
- **Rate Change** devices control the rate at which the node will transmit data, and this can be at a fixed rate or a variable rate. Rate change devices may make decisions based on SNR (Signal to Noise Ratio) information provided by the modem, or by examining the detected FER (Frame Error Rate).
- **Transmission Control** defines the pattern of data transmission. The three basic types of transmission are *simplex*, *duplex* and *broadcast*. Transmission Control also controls if ALE is used.
- **EOT Engine** is an analysis engine which handles converting received EOT values into the best available estimate of the actual end-transmission time according to different methods.
- **Rx-end Handler** driver checks, fixes up or synthesizes rx-end events, which is the event that is generated by some modems when they have completely finished decoding the data of a transmission. This device is not usually needed.
- **ALE Device** driver specifies the ALE service to use. At present only the MoRaSky simulated ALE device is supported.
- **ALE Address Mapper** controls how addresses are mapped when an ALE device is configured. This must be configured when ALE is used.

At a minimum the following devices must be configure a working basic node configuration:

- **Modem** Whether a single modem, a transmit modem only (if the node only transmits), a receive modem only (if the node receives data only) or both transmit and receive modems.
- **Rate Change** Whether fixed rate or variable (variable rate change devices require some form of control connection to the modem).
- **Transmission Control** In order to control when the node can transmit or receive (depending on simplex, duplex modes, etc.).

### 3.7.1 Driver Respawning Configuration

In the event that a driver fails it may be restarted (respawned). This configuration handles the logic used to perform respawning. This can be configured independently for each driver.

Use of default configuration is recommended. These settings are most likely useful for those developing custom drivers. Configuration options:

- **delay** Delay before respawning, default 2s.
- **limit** Limit to number of respawns that are permitted recently before respawning is abandoned, default infinite.
- **cutoff** Time period (in seconds) which counts as ‘recent’ for applying the conf.limit, default 60s
- **max_mem** Maximum VM memory in KB, or 0 to disable checking, default 10MB.
- **max_cb** Maximum VM callback count, or 0 to disable checking, default 1000.

Each time the component fails and respawning considered, there are two options:

- Restart it, after a delay (delay).
- Raise an error in this component and propagate the failure upwards. This is done if more than limit failures occur in cutoff seconds. By default no error propagation is done.

## 3.7.2 Modem Device Types and Configuration

This section describes modem configuration options.

### 3.7.2.1 One and Two Modem Setup

Icon-5066 nodes can be configured with either a single modem or a pair of modems (Rx modem and Tx modem). Single modem configuration is used in three scenarios:

- Simplex operation where a single modem will alternately transmit and receive data.
- Broadcast operation (send or receive) where data is sent in one direction only.
- Single modem split site operation where a single modem is used, with connections to two radios (one for transmit and one for receive). This can be used for duplex or simplex operation.

Icon-5066 can also be configured for dual modem split site operation, where separate modems are used for transmit and receive.

### 3.7.2.2 Common Modem Configuration

This section covers configuration that is common to all modem device drivers.

#### 3.7.2.2.1 Streaming Configuration

Each modem device layers streaming and a transmission queue on top of the serial driver or data stream interface.

The default value of clock is recommended for half-duplex communication, but risks underflow or overflow for broadcast or duplex. The streaming configuration may be tuned to reduce latency. The possible values, each shown with valid additional options, are:

```
clock [moredata=secs] [level=secs] [blocks=cnt]
```

- Request a pre-fill amount of data initially then ask for more data a block (or groups of blocks) at a time according to bps rate, without any feedback from modem.
- Advantage: Buffer fill levels strictly controlled by driver.
- Disadvantage: Risk of clock drift for broadcast/duplex leading to underflow or overflow.

```
max [moredata=secs] [refill=secs]
```

- Use serial device’s flow control indications to keep TX buffer full.
- Advantage: No clock drift.

- Disadvantage: Buffers in chain to modem may be large, especially compared to BPS rate.

```
min [moredata=secs] [level=secs] [blocks=cnt]
```

- Use modem's buffer-status command, if available, to decide when to send more data.
- Advantage: No clock drift
- Advantage: Keeps buffers lean

The max streamer is recommended to be used for anything with tight flow-control (for example the MIL-STD-188-110D Appendix A TCP protocol used by the Rockwell driver). Configuration sub-options are:

- moredata=seconds: Time to allow for Icon-5066 core to produce more data when requested (default 2s)
- refill=seconds: How often to refill for max, default 2s. This is really the gap between HWM and LWM.
- level=seconds: Minimum preload level in seconds of data for clock, minimum target level for min. Default is 5s.
- blocks=count: Minimum preload level in blocks of data for clock, minimum target level for min. Default is 2 blocks.

Note on crypto devices: Those that add a header to the data stream will make the block-alignment calculations of clock incorrect. However, they will only cause a small amount of extra data to be prefetched, so this is not expected to cause problems.

#### 3.7.2.2.2 Serial and TCP Driver Configuration

Serial and standard TCP data driver configuration is configured by the "Serial Driver Configuration" option, which is common to all modem drivers, including the generic modem. There are four types of serial driver supported:

- Synchronous Serial, using SeaLevel card with Windows drivers. This is appropriate for use with crypto.
- TCP using MIL-STD-188-110D Appendix A. This is the open standard for TCP connection to HF modems. This can be used with any modem that supports this standard. All of the supported Rockwell Collins modems support this standard.
- Asynchronous Serial, using Windows COM ports. This may be useful for operation without crypto where modems do not support TCP data, such as RapidM RM8.
- MoRaSky can simulate serial line interface to modems. This may be useful for testing.

Drivers for MIL-STD-188-110D and MoRaSky do not need any setup.

For Windows COM port, an appropriate COM port needs to be configured to associate with the serial driver. It is recommended to use a speed somewhat higher than the fastest modem speed anticipated.

For Synchronous Serial the Sealevel "Seamac V6" Windows 64-bit drivers need to be downloaded and installed. This is available from the SeaLevel web site.

The Serial Driver Configuration is specified in this version of Icon-5066 with a structured string. GUI configuration is planned for future version. Typical example configurations are shown for each of the four driver types:

- Synchronous Serial e.g.:

```
driver=serialproxy open{exe=seamac_win dev=SeaMac0}
port{polarity=normal if=rs232}
```

- MIL-STD-188-110D Appendix A e.g.:

```
driver=rc_data open{host=192.168.130.12 port=3000}
```

- Asynchronous Serial e.g.:

```
driver=serialproxy open{exe=stdserial dev=COM1} port{baud=9600
data=8 parity=N stop=1 rtscts=oneway drain_ms=200}
```

- MoRaSky e.g.:

```
driver=serialproxy open{exe=morasky}
```

At the top level, there are two drivers: **rc_data** for MIL-STA-188-110D and **serialproxy** for everything else.

The **rc_data** driver has an open parameter, with host and port sub-parameters that define where to connect to.

The **serialproxy** driver has two first level parameters (**open** and **port**) each with sub-parameters. The **open** parameter has two sub-parameters:

- **exe** Name of the driver. Choices:
  - `stdserial` For Windows asynchronous serial.
  - `seamac_win` For SeaLevel synchronous serial (Windows).
  - `seamac` For SeaLevel synchronous serial (Linux).
  - `morasky` For MoRaSky.
- **dev** Port to open. The port name depends on the platform and whether asynchronous or synchronous serial. For asynchronous serial on Windows an example is `COM1`. For synchronous serial with the SeaMac driver the port is the SeaLevel instance name, e.g. `SeaMac0`.

The **port** parameter has these sub-parameters in the *asynchronous* serial case:

- **baud** Asynchronous baud rate, which can have values from `75` to `115200`. The default is `9600`.
- **data** Data bits 5, 6, 7 or 8. The default is 8.
- **parity** Asynchronous parity bit O, E, N, M or S (odd/even/none/mark/space). The default is N (none).
- **stop** Asynchronous stop bits 1 or 2. The default is 1.
- **rtscts** RTS/CTS handling mode. Options are:
  - `disable` RTS and DTR off, no CTS handling.
  - `on` RTS and DTR locked on permanently.
  - `oneway` TX flow control only (the original meaning of RTS/CTS, aka `toggle` on Windows).
  - `transmit` RTS locked on while transmitting, wait for CTS to drop at end.
  - `twoway` Use RTS as RTR for two-way flow control (aka `handshake` on Windows).
- **drain_ms**Time to allow for hidden buffers to drain, in milliseconds. If a serial device has hidden buffers that are not visible to the driver through the OS calls, then the driver will drop RTS too early. When RTS is dropped, any data still being transmitted from the hidden buffers will be lost. Configure a drain time here to allow time for these buffers to clear. The default is 0.

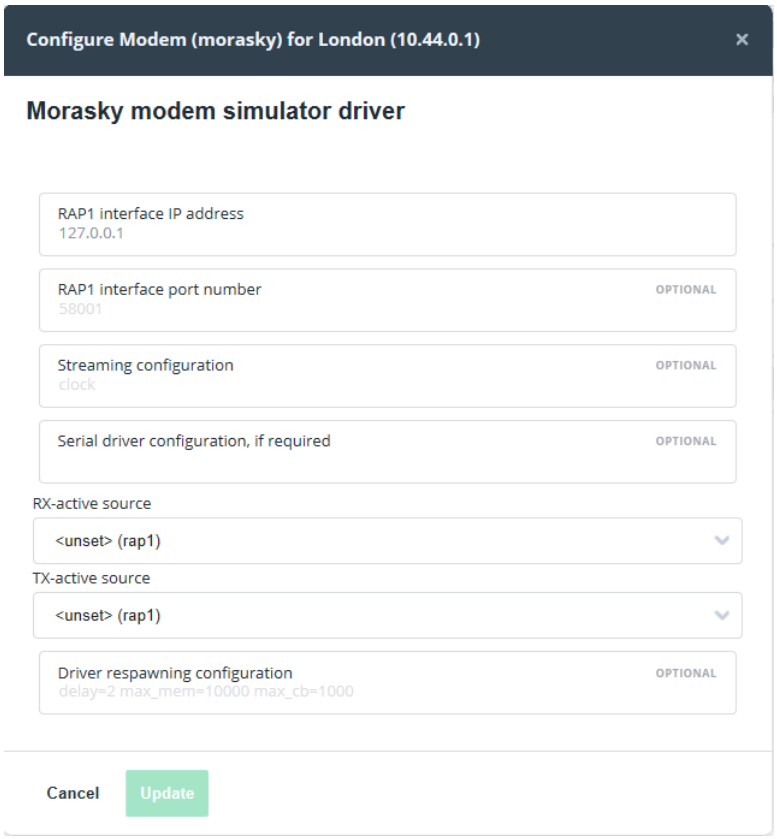
The **port** parameter has these sub-parameters in the *synchronous* serial case:

- **if** Electrical interface type. Possible values are `rs232`, `rs422` or `rs423`. The default is `rs232`.
- **polarity** Signal polarity. Possible values are `normal` or `inverted`. The default is `normal`.

- **tx_clock** TX clock. Possible values are `tx`, `rx`, `dpll` or BPA rate as a number. The default is `tx`.
- **rx_clock** RX clock. Possible values are `rx`, `tx`, `dpll` or BPA rate as a number. The default is `rx`.

### 3.7.2.3 MoRaSky Modem

**Figure 3.7. MoRaSky device configuration**




MoRaSky is Isode's software modem and channel simulator that can be used to test Icon-5066 in the absence of a hardware modem. MoRaSky uses the RapidM RAP1 modem protocol. Modem device specific options are:

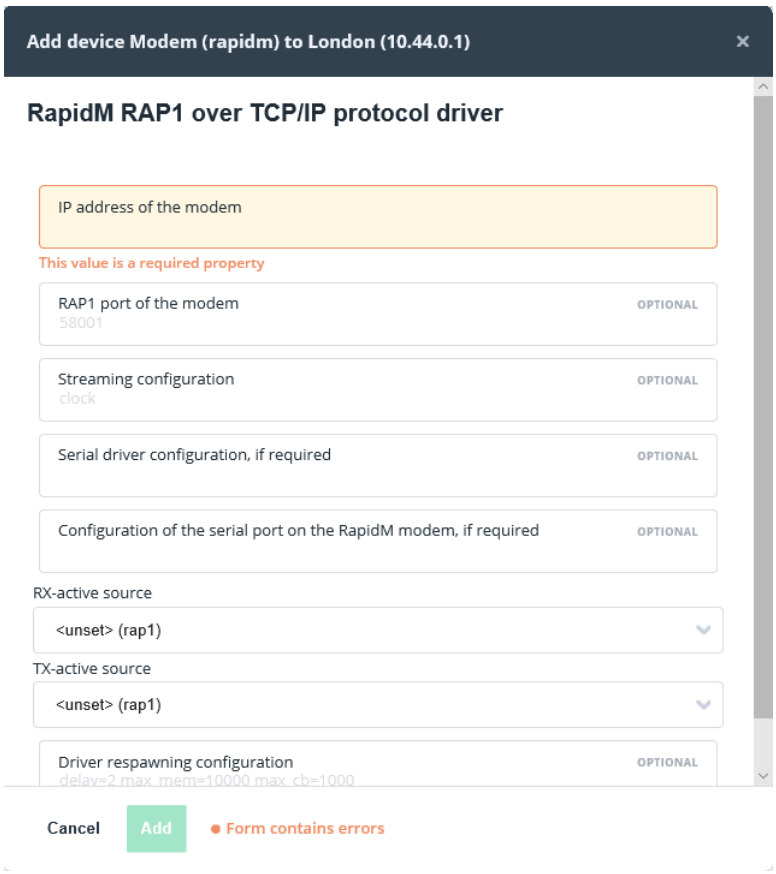
- **RAP1 interface IP address** The IP address to connect to MoRaSky. This is a mandatory configuration option.
- **RAP1 interface port number** The port used by the MoRaSky. Default is 58001.
- **RX-active source** RX-active (carrier) information may be sourced either over the control link (RAP1), or from the serial device. This is not generally needed.
- **TX-active source** TX-active (transmitting) information may be sourced either over the control link (RAP1), or from the serial device.

Common configuration options:

- **Streaming configuration** See Section 3.7.2.2.1, “Streaming Configuration”. This is not generally needed.
- **Serial driver configuration** See Section 3.7.2.2.2, “Serial and TCP Driver Configuration”. If this is not specified then the driver defaults to data over RAP1.
- **Driver respawning configuration** See Section 3.7.1, “Driver Respawning Configuration”.

### 3.7.2.4 RapidM Modem

**Figure 3.8. RapidM modem configuration**




The **rapidm** driver supports RapidM modems using the RAP1 control interface. Driver specific options are:

- **RAP1 interface IP address** The IP address to connect to the modem. This is a mandatory configuration option.
- **RAP1 interface port number** The port used by the modem. Default is 58001.
- **Configuration of the serial port on the RapidM modem, if required** If serial communication is used with the RapidM modem, these parameters are sent over RAP1 to configure the RapidM end of the serial connection. This can be kept blank in order to use the modem's existing on device configuration. These parameters should match the serial driver. For example “mode=async baud=9600 parity=N stop=1 rtscts=1 elect=rs232”. Options are:
  - **mode**:
    - *async* Asynchronous, sending start/stop over radio.
    - *sync* Synchronous serial.
    - *hs-sync* Asynchronous, send just data over the radio. This is the default.
  - **inverted** Inverted polarity: 1 on, 0 off. Default is 0.
  - **elect** Electrical mode: *rs232*, *rs422* or *rs423*, default is *rs232*.
  - **clock** Synchronous serial clock source: I, O or R (input/output/recovered), default is O.
  - **baud** Asynchronous baud rate 75 to 115200. The default is default 9600.
  - stop Asynchronous stop bits: 1 or 2, default 1.
  - **parity** Asynchronous parity bit: O, E or N (odd/even/none), default N.
  - **rtscts** Asynchronous RTS/CTS: 1 on, 0 off, default 1 (on).

- **dtrdsr** DTR/DSR handshaking: 1 enable, 0 disable, default disable.

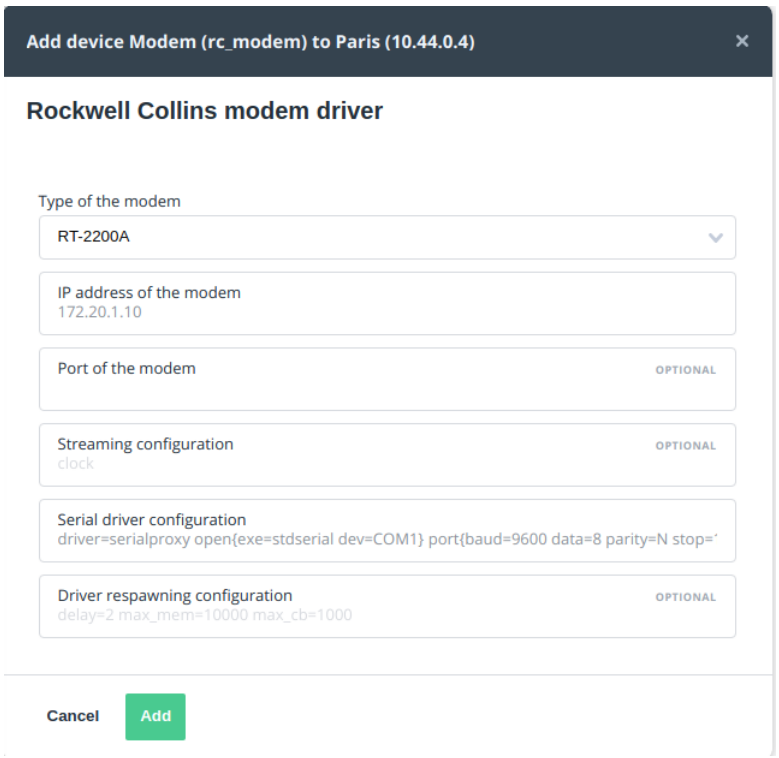
- **RX-active source** RX-active (carrier) information may be sourced either over the control link (RAP1), or from the serial device.
- **TX-active source** TX-active (transmitting) information may be sourced either over the control link (RAP1), or from the serial device. .

Common configuration options:

- **Streaming configuration** See Section 3.7.2.2.1, “Streaming Configuration”.
- **Serial driver configuration** See Section 3.7.2.2.2, “Serial and TCP Driver Configuration”. This must be specified for RM8 modems. For RM6 modems, this is normally omitted, as data will be sent and received over the TCP control channel. It may however be specified in the case that synchronous serial is in use.
- **Driver respawning configuration** See Section 3.7.1, “Driver Respawning Configuration”.

### 3.7.2.5 Rockwell-Collins Modem

**Figure 3.9. Rockwell-Collins modem configuration**




The **rockwell_collins** driver supports Rockwell-Collins using the MIL-STD-188-110C Appendix A modem interface definition for data. This will be used unless serial driver configuration is specified. The following specific modem models are supported:

- HSM-2050
- RT-4800
- RT-2200A
- Q9600
- Q9604

The following options are mandatory:

- **Type of the modem** The specific modem model (a choice from the list provided above).
- **IP address of the modem** The IP address of the modem for control purposes.

- **Port of the modem** The network port of the modem.

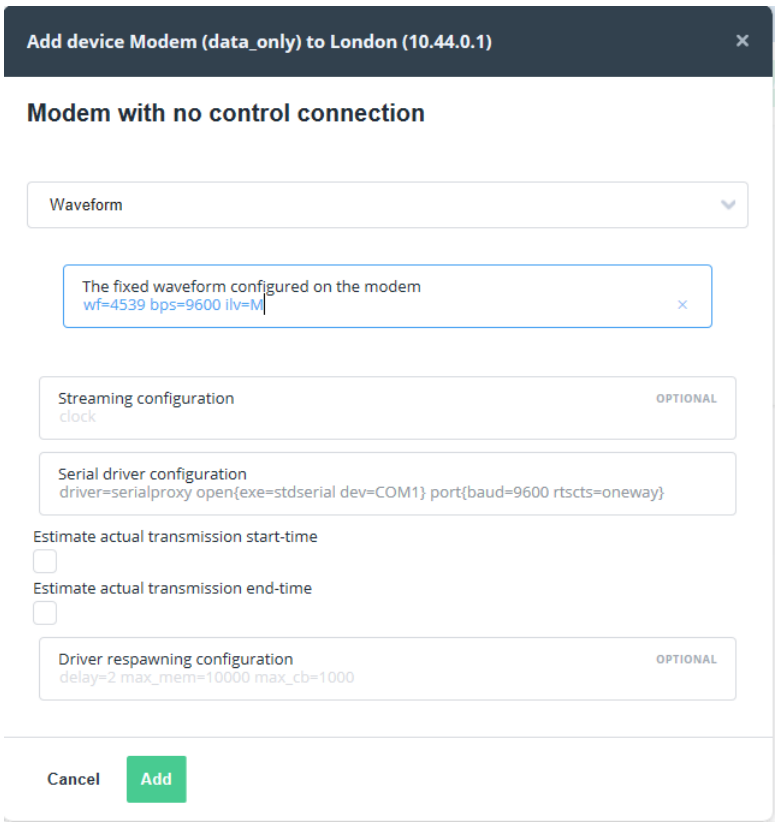
Common configuration options:

- **Streaming configuration** See Section 3.7.2.2.1, “Streaming Configuration”.
- **Serial driver configuration** See Section 3.7.2.2.2, “Serial and TCP Driver Configuration”.
- **Driver respawning configuration** See Section 3.7.1, “Driver Respawning Configuration”.

### 3.7.2.6 Generic Modem

The **data_only** driver is a generic driver for communication with a modem at fixed speed over a serial connection. This allows connecting to a modem that only has a data connection and where it is not possible to change the waveform on the modem or access channel quality information. An appropriate serial driver must be configured. Either the waveform or the BPS rate configured on the modem must be provided.

**Figure 3.10. Generic modem configuration**

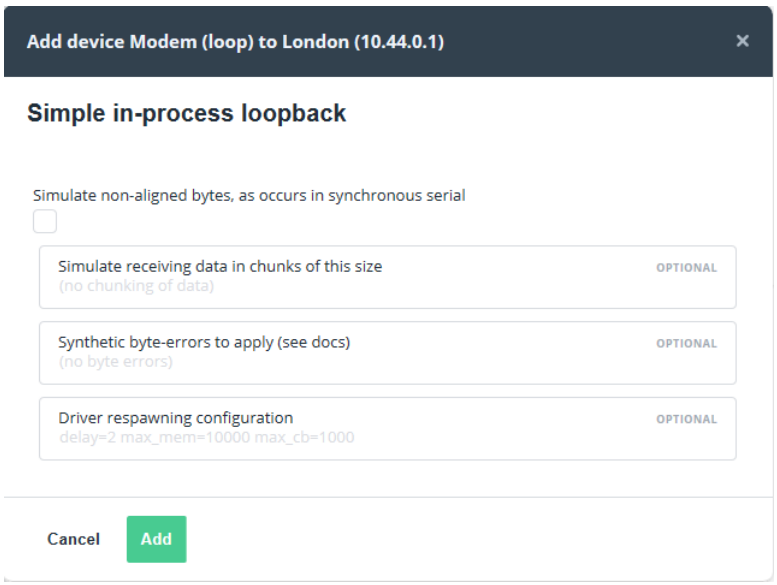



The set of options vary depending on the information available from the modem, and specifically whether the **Waveform** or **Without Waveform** selection is chosen:

- **Waveform** Use this choice when the modem waveform configuration is available. In this case the following options are presented:
  - **The fixed waveform configured on the modem**
- **Without Waveform** Use this choice when the modem waveform configuration is not known. Options in this case are:
  - **Configured waveform data rate in bits-per-second** e.g. 9600, 19200, etc.
  - **Configured waveform interleaver block size in bits** Default is 64.
  - **Configured waveform padding in bytes** Padding bytes are inserted by the modem or crypto devices. Normally this would be at least 4 for the EOM mark. Default is 4.

#### 3.7.2.7 Loopback Modem

**Figure 3.11. Loopback device configuration**




A loopback modem can be configured so that traffic comes straight back. This is used for specialized testing and is not generally used.

### 3.7.3 Rate Change Devices

A Rate Change driver is always needed. There are three options:

- **SNR Based** Measures the SNR (Signal to Noise Ratio) on the receiving system to help select the best parameters. This option is recommended for most configurations.
- **Fixed Speed** This can be useful for some deployments.
- **FER Based** This controls speed based on FER (Frame Error Rate). It is typically used with modems that do not report SNR.

The following sections describe the configuration of each option.

#### 3.7.3.1 SNR Based Rate Selection

The **snr** driver is a simple rate-change driver based on SNR measurement. For each SNR range one particular speed is judged the best one to transmit at by table lookup (a generic table is provided).

The driver follows Isode's S5066-EP4 (Data Rate Selection) specification, and here the sender will use two settings based on data to send:

- A speed chosen to optimize throughput. Some data loss and retransmission is anticipated. This is chosen for bulk data such as messaging; or
- A speed chosen to minimize data loss and optimize for low latency. This is chosen for acknowledgements and applications such as instant messaging.

The SNR based rate change device option has the following configuration options:

- **EOW11 message to send** If omitted, then no EOW 11 messages are sent, forcing the remote end to use a default. Use 8 to specifically request EOW 8/12, or 0 for just EOW 8. To include both bulk and low-latency waveforms, add 4, e.g. 12 gives EOW 8/9/12. EOW 12 is not required for correct operation of this SNR-based rate-change driver, but it may be useful in case the modem driver doesn't detect the incoming BPS rate. If no value is set and the remote end doesn't use a sensible default then interoperability issues may arise.

- **EOW11 message to assume if none received** Default of 12 means to send EOW 8, 9 and 12.
- **Waveform family** This gives a drop-down selection of the auto-baud waveform to use. Choices:
  - **STANAG 4539**
  - **STANAG 5069 (WBHF)** When this is chosen, the channel bandwidth (3kHz to 48 kHz) must also be chosen as part of the option.
- **SNR lookup table** This allows selection of a default lookup table. The choices are:
  - **4539 Groundwave** Suitable for STANAG 4539 with Groundwave.
  - **4539 Skywave** Suitable for STANAG 4539 with Skywave.
  - **5069 Groundwave** Suitable for STANAG 5069 with Groundwave.
  - **5069 Skywave** Suitable for STANAG 5069 with Skywave. These values are chosen based on STANAG 5069 specifications to make a reasonable balance between CCIR POOR, CCI MEDIUM and CCIR GOOD. This is a sensible general purpose Skywave setting.
  - **5069 Skywave Base** Suitable for STANAG 5069 with Skywave. These values are chosen based on STANAG 5069 specifications for CCIR POOR. This is a sensible choice when conditions are known to be poor.
  - **CUSTOM** Use the configured custom table.
- **Custom SNR Lookup Table** This allows definition of custom behaviour. This is a space separated string, with values as follows:
  - **Interleaver offset** This gives an SNR offset for chosen interleaver. For example, US=10 means that 10 dB is added to the SNR before comparison with the table. This enables a generic mechanism to shift the transition points based on interleaver, as shorter interleavers will need slower speeds for Skywave.
  - **STANAG 4539 speeds** This defines the minimum SNR needed to use a specified speed. For example “b6400=24” means that a minimum SNR of 24 is needed to transmit at 6400 bps. Specific settings for a given interleaver can be made, which will override the interleaver offset. For example “b1200L=11”.
  - **STANAG 5069 Waveform** This gives a minimum SNR for a given waveform, For example “w2=5” means that a minimum SNR of 5 is needed to use waveform 2. Separate values can be defined for specific bandwidths. For example “w3b9=8” means that minimum SNR for waveform 3 at 9kHz is 8. These can be modified by interleaver. For example “w2L=4” or “w3b9US=10”.
- **SNR Adjust in dB** This gives a simple way to make the rate change more aggressive or more conservative, by simply adjusting the change points. A positive value will make the rate change more aggressive (choosing faster speed) and a negative value will make the rate change more conservative (choosing lower speeds).
- **Initial BPS** This is the speed that will be used for first transmission if there is no additional information on the choice. If CAS-1 link fails to establish retries will aggressively drop speed from this setting. Default is 300 bps, which is the value required by C.6.4.1 of STANAG 5066. A faster default speed may be better in some scenarios.
- **Reset time in minutes** Icon-5066 determines the best speed to be used for bulk and low latency transmissions. These values are remembered for a period and will be used during that period. After this period during which no updates are received from the peer, configured by this setting, the values will not be used and Initial BPS will be used instead. The default setting is 30 minutes.
- **Bulk data interleaver** This is the interleaver used for bulk data transfer. If the chosen setting is not available in the active waveform/speed, the closest match will be used. Default is VL. A long interleaver is recommended to minimise data loss.
- **Low latency data interleaver** This is the interleaver used for low latency (minimise data loss) data transfer. If the chosen setting is not available in the active waveform/speed, the closest match will be used. Default is VS. These transmissions will generally be short, so a long interleaver is not recommended.

- **Low latency maximum transmission duration** The decision to use bulk or low latency settings is based on the amount of data to send. The (slower) low latency speed is used if all data can be transmitted in this time. Otherwise the bulk speed is used. Default 5 seconds.
- **CPDU Segment Size mapping** The maximum C_PDU Segment Size is chosen based on selected transmission speed. This is specified by syntax speed=length with multiple values space separated. If a value is not specified for the chosen speed, the length will be interpolated using the nearest speed value or values.
- **Number of times to repeat ACK packets** If an ACK packet requesting retransmission is lost, this will increase latency for the referenced data. As ACK packets are small, repeating them has low overhead and guards against this. This setting specifies the number of times to send each ACK. 1 means send the ACK once and do not repeat. Default is 4.
- **Repeat counts for D_PDUs sent at least 1/2/3 times** When transmitting bulk data significant (20-50%) FER is likely. This makes it likely that some ARQ D_PDUs will be retransmitted at least 3 or 4 times. ARQ applications often use "in order" delivery, so that one delayed D_PDU will effectively block all subsequent D_PDUs. These options allow additional repeats transmissions to be made, and enable the number of repeats to be configured based on the number of transmissions made. This will improve latency for bulk applications at the expense of some throughput. Default setting for each is 1, which means only send once and do not repeat.
- **Maximum transmission time** in seconds This is the longest transmission allowed, apart from duplex and broadcast circuits. Default is 127.5 seconds, which is the maximum time allowed in the standard. A shorter time may be chosen to ensure better latency, but this may lead to reduced throughput.
- **Minimum transmission time** in seconds This enables forcing a minimum transmission time. Where data can be transferred in less time, the extra transmission time will be used to duplicate data. This option may be used to protect against very short transmissions getting lost in fades. Default is 0, which means no minimum length.

#### 3.7.3.2 Fixed Speed

The **fixed** driver does not monitor channel quality and does not perform any rate-change. Instead it simply applies a fixed set of parameters. This driver is useful in some situations, particularly where crypto bypass is not permitted. The values are as follows:

- **Waveform** Drop down selection of Waveform, Speed, and Interleaver to be used.
- **Repeat count for DPDUs sent at least 1/2/3 times** When transmitting bulk data significant (20-50%) FER is likely. This makes it likely that some ARQ D_PDUs will be retransmitted at least 3 or 4 times. ARQ applications sometimes use "in order" delivery, so that one delayed D_PDU will effectively block all subsequent D_PDUs. These options allow additional repeats transmissions to be made and enable the number of repeats to be configured based on the number of transmissions made. Setting these values to higher than 1 will improve latency for bulk applications at the expense of some throughput. Default setting for each is 1, which means only send once and do not repeat.
- **Maximum transmission time in seconds** This is the longest transmission allowed, apart from duplex and broadcast circuits. Default is 128 seconds, which is the maximum time allowed in the standard. A shorter time may be chosen to ensure better latency, but this may lead to reduced throughput.
- **Minimum transmission time in seconds** This enables forcing a minimum transmission time. Where data can be transferred in less time, the extra transmission time will be used to duplicate data. This option may be used to protect against very short transmissions getting lost in fades. Default is 0, which means no minimum length.
- **CPDU Segment Size mapping** The maximum C_PDU Segment Size is chosen based on selected transmission speed. This is specified by syntax speed=length with multiple values space separated. If a value is not specified for the chosen speed, the length will be interpolated using the nearest speed value or values.

- **Number of times to repeat ACK packets** If an ACK packet requesting retransmission is lost, this will increase latency for the referenced data. As ACK packets are small, repeating them has low overhead and guards against this. This setting specifies the number of times to send each ACK. 1 means send the ACK once and do not repeat. Default 4.
- **For interval-based duplicates list of intervals** This is for use with broadcast, where stations are unable to ACK or NACK reception, so DPDUs should be repeated at intervals to try and ensure reception. The values is comma-separated list of intervals in seconds. Examples: "10,10,10" and "10,20,30".

#### 3.7.3.3 FER Based Rate Selection

This is an implementation of the Trinder-Gillespie frame-error-rate approach to rate-change for S'4539 waveforms. This could be used as a basis for other FER-based rate-change drivers using a different algorithm. This driver is useful if the modem does not provide SNR information. If no EOW 11 config options are set, then they default to values that will work with the same driver at the remote end. Configuration options are all described in SNR configuration.

### 3.7.4 Transmission Control

This device type controls the mode and pattern of data transmission to the modem. There are three basic types of transmission:

- Simplex, where a single modem/radio is used to both send and receive data. For simplex, transmissions are limited to a maximum of 127.5 seconds and EOT (End of Transmission) is used with each D_PDU to indicate when a transmission will finish.
- Broadcast, where data is sent in one direction only. EOT is not set and there is no limit on transmission length.
- Duplex, where two radios and one or two modems are used to send data in both directions at the same time. EOT is not set and there is no limit on transmission length.

Data transmission options allow for "gaps" and "continuous"", which controls what happens when there is no data to send. For "gaps", transmission is finished when there is no more data to send. For "continuous" the system keeps sending over the air, even when there is no application data to send.

The available device choices are listed below:

- **si_gaps** This is simplex transmission, with transmission ending when there is no data to send. This should always be used for multi-node systems.
- **si_cont** This is simplex transmission. When a node has no data to send, it will simply transmit padding data. If there is a CAS-1 link open, it will pass control to the other end of the link. This is appropriate for a two-node network where it is desired to keep the channel active. It can be more responsive to arriving data.
- **bc_gaps** This is for duplex or broadcast transmission. When there is no data to send, transmission ceases.
- **bc_cont** This is for duplex or broadcast transmission. When there is no data to send, padding data is sent. This mode is recommended for duplex.

### 3.7.5 EOT Engine

The algorithm used to determine how to analyse EOT (End of Transmission) information is configurable, and one of a number of drivers may be selected. Configuring this driver allows selection of non-default behaviour.

- **bps** Adjusted EOT handling, for a block-based data interface, using BPS from modem. If EOT values in the DPDU headers are calculated according to S5066, but data is transferred from the modem in interleaver block units, then the transmission end-time

estimate will be poor. It is necessary to adjust the estimates according to how far back the DPDU header is in the received data block by using the BPS rate. This driver assumes that the incoming waveform's BPS rate will be provided by the rate-change driver or the modem driver. If the BPS is not available, then the end-time estimate is likely to be an overestimate, the same as if the 'raw' driver were used.

- **bps_est** Adjusted EOT handling, for block-based data interface, using BPS estimates. If EOT values in the DPDU headers are calculated according to S5066, but data is transferred from the modem in interleaver block units, then the transmission end-time estimate will be poor. It is necessary to adjust the estimates according to how far back the DPDU header is in the received data block by using the BPS rate. This driver falls back on measurements of the incoming waveform BPS in case the BPS is not provided by the rate-change driver or the modem driver.
- **none** No EOT handling, for duplex/broadcast/receive-only. This is the default for duplex/broadcast transmission control.
- **raw** Unprocessed EOT handling, for serial connection at modem rate. This uses each EOT value as it arrives. For data arriving over serial at the modem rate, this will give the correct EOT calculation. The 'raw' choice is also used when receiving data over TCP/IP where EOT has been set the same for the whole block by the remote end. This is the default for simplex transmission control.

If no EOT Engine device is set, the default algorithm is used with default parameters. The default is recommended.

### 3.7.6 RX-end Handler

This driver can be configured when behaviour other than the default is desired. This specifies the algorithm used to determine when a receive transmission is considered to be terminated. This may be configured to give faster switching (to improve performance) or slower switching (to give devices sufficient time to switch). One of a number of drivers may be selected:

- **carrier** Generates RX-end based on carrier drop and incoming data. Uses carrier drop and incoming data to judge when to report RX-end. With defaults, or with a small delay, this may work well for serial drivers that don't send RX-end. For situations where it is not known how long data may come in after carrier drop, setting delay to a value longer than the interval between rx-data events will take care of it. The chunks can be used to handle that situations where there are always one or more rx-data events following carrier drop.
- **check** Pass through RX-end events, but check for problems. Checks for these errors: missing RX-end events, RX-end after EOT expiry and rx-data after RX-end.
- **data** Generates RX-end based solely on incoming data. Uses incoming data to judge when to report RX-end. If there is no carrier information reported at all by the device, then this may be the only way to judge the end of the transmission.
- **pass** Pass through RX-end events, without checks. This is suitable for duplex and receive-only where the checks aren't valid.

If no RX-end device is used, the default algorithm is used with default parameters. The default is check recommended.

### 3.7.7 ALE

Automatic Link Establishment (ALE) is an important capability of modern HF networks. Icon-5066 provides a framework so that ALE can be used to establish links between a pair of HF Nodes.

The ALE protocol is provided by drivers for the specific ALE implementation, which may be 2G, 3G or 4G ALE. This allows for any ALE implementation to be used with Icon-5066. The current Icon-5066 release does not provide support for any real ALE drivers. However,

it does provide support for a MoRaSky driver, so that the Icon-5066 ALE capability can be demonstrated.

In order to configure ALE for a node, the first step is to select the Transmission Control option of **ale_121**. This mode reflects use of point to point ALE (1:1). This selection enables use of ALE for the node. It is anticipated that future releases will provide options for multi-node ALE, so that ALE can be used in conjunction with Annex K or Annex L.

The **ale_121** options are shown below:

- **Seconds to wait before terminating ALE link** Default 20. Peer response lost or other error. Return to scanning.
- **Number of empty (padding) transmissions to make before terminating ALE link** If there is no data received, it may be worthwhile to send data back and forth just in case. Default 0.
- **Seconds to wait for modem to drop RX-active after EOT** This is to deal with unexpected modem behaviour or third party signal. Default 10.
- **Seconds to wait before aborting transmission (both Rx and Tx)** This is to deal with unexpected modem behaviour. Default 150.

When ALE is configured for a node, it is recommended to select **implicit soft links** in the configuration. This will save the overhead of explicitly negotiating a CAS-1 soft link, as this is not needed when ALE is used.

When ALE is used, a ALE Device driver must be selected. Currently, the only option is the morasky driver, which will use the MoRaSky simulator. The only MoRaSky driver option is the standard respawning configuration.

MoRaSky does not need any specific options to run with ALE. It will assign ALE addresses to each node of R1, R2, R3, R4, etc..

When ALE is used, and ALE Address Mapper driver must be used. There are two options:

- **pass** This driver simply passes the STANAG 5066 Address of the peer to the ALE driver.
- **local** This uses a configured mapping of STANAG 5066 Address of the peer to an ALE address. This is the option the morasky ALE driver needs.

The **local** driver has a single parameter of Mapping Table. This is encoded as a string with space separated values, with each value defining a mapping, for example `10.88.0.1=R1 10.88.0.2=R2 10.88.0.3=R3`.

# Chapter 4 Icon-5066 Monitoring

This chapter describes the monitoring view, which displays the status of each configured S5066 node.

The monitoring view (shown by clicking on the **Status** icon in the sidebar), like the configuration view, shows a number of panels where each panel corresponds to one of the configured S5066 nodes.

**Figure 4.1. Monitoring screen**


Each panel is labelled by the configured node name and S5066 address, and displays node status in a number of sub-panels:

- **CAS-1 Link Status** That status of any CAS-1 links to other nodes. Links may be established, in process of establishment or broken.
- **Modem** Current transmission state (RX or TX), transmission parameters (speed, waveform, etc.) and the SNR as reported by the modem.
- **Tx Status** State of current transmission or time since last transmission.
- **Rx Status** State of current reception or time since last reception.

# Chapter 5 Analysis and Debug

This chapter looks at how to perform detailed analysis and debug for Icon-5066.

## 5.1 Event Storage

Events are aggregated in and resulting logs written by the `isode.ddsd` Distributed Data Service. On Windows systems logs are typically written to:

*C:\Isode\log*

The path above can be changed at Windows package installation time. On Linux logs are written to:

*/var/isode/log*

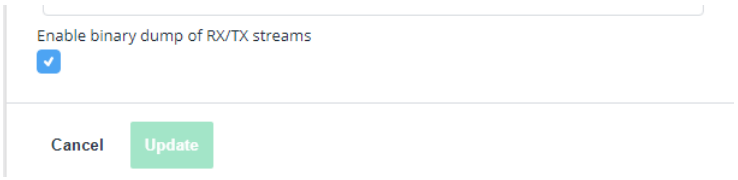
Logging is written on a per-session basis. Specifically, a new directory is created for each run of the **isode.s5066d** Core Protocol Server. The per-session directory is named by a unique sessions that is constructed from the date and time of the start of the run and the process ID of the server, for example *20180815-104814-5252*. A number of objects are written to the log directory for each session, and these are:

- *Configuration file* The *icon5066.json* configuration file is archived for each session, in order to be able to tie any log analysis back to the configuration used in the run.
- *Event log* The *icon5066-event.log* event log file contains notifications of server level events (e.g. server start, stop, failure to connect to a modem, etc.).
- *History database* The *tsdb* directory contains a trace for each configured STANAG-5066 node. The history database file corresponding to each node is named by the STANAG-5066 node address, e.g. *10.44.0.1*. The contents of these files is not human readable, and is intended to be processed by analysis tools only.
- *Process trace* A trace file is created at server startup and is used to store a stack trace of the protocol server in the event of a server crash. The file is named by a fixed string and the process ID of the `isode.s5066d` process, e.g. *s5066d.2796.trace.log*.

## 5.2 Modem Communication Tracing

In some situations the default logging may not contain enough information to permit diagnosis of a particular problem. In this case the raw STANAG-5066 data stream between the server and the modem can be dumped. The raw streams can then be converted to the *pcap* packet capture format and viewed in third-party tools such as Wireshark. The steps required to enable this for a particular node are:

1. Enable binary dump of RX/TX streams in the per-node configuration dialog as below:




This step will result a trace file named by the node address in with a suffix of *.bin* to be written to the log directory.

2. Convert the raw binary data file to *pcap* using the **bin2pcap** utility. By default **bin2pcap** will look for any and all binary data files (files with suffix *.bin*) in the current working directory and convert them to the *pcap* format.

# Chapter 6 MoRaSky: Modem, Radio and Sky simulation

This chapter explains how to use the MoRaSky application to simulate serial devices, radio modems, crypto boxes, radios, antennae and the transmission channel, and various scenarios of bit errors and interference affecting that system.

## 6.1 Target Audience

MoRaSky is useful for offline testing of any application that needs to communicate over an HF radio channel, for example for testing Icon-5066 or an ACP-127 channel whilst introducing controlled errors. It also finds a use when designing rate-change drivers to test how the driver behaves under various simulated scenarios.

## 6.2 Introduction

MoRaSky is a Java application that acts as one or more modems listening on local ports, talking RAP1 (RapidM) or RC-data (Rockwell-Collins) over RIPC over TCP/IP. The protocol used (RAP1 or RC-data) is established using protocol detection. A virtual serial hub is also simulated, accessible over TCP/IP, with an output connected to the data serial port of each modem. Each modem drives a virtual radio and antenna and signal propagation between the antennas is simulated.

The following aspects are all simulated, to varying degrees:

- Both synchronous and asynchronous serial behaviour and flow control in the serial hub.
- Waveform timings (for Stanag 4285/4539 and MIL-STD-188-110C WBHF)
- Modem timings
- Crypto boxes, including the varying sensitivity to bit errors in the crypto header and data body
- Propagation delays
- Collision of multiple simultaneous transmissions on the same frequency
- Bit error generation scenarios, including bit error clustering, padding with junk and dropping part of the transmission
- SNR-based bit error generation based on table lookup
- Abruptly changing interference
- Externally-controlled channel simulation; this is configurable remotely by **hftool** for extended tests

## 6.3 MoRaSky Tool Usage

For UNIX, the **morasky** command can be found in *(SBINDIR)*. It must be run from a terminal window. Use Ctrl-C to stop it.

For Windows, a few sample MoRaSky invocations are found in the Isode menu. These bring up a terminal window and run MoRaSky. Use Ctrl-C in that window to stop the simulation. To generate a custom MoRaSky configuration, right-click and copy one of the MoRaSky invocations from the menu, and then right-click and paste it onto the desktop. Then right-click Properties allows the shortcut to be renamed and the MoRaSky command-line to be modified to add the options explained below. This shortcut can then be used to start a MoRaSky session with that configuration. In this way it is possible to set up several different shortcuts for different configurations.

To communicate with MoRaSky from Icon 5066 and **hftool** the `morasky` modem driver should be used, which uses the RAP1 protocol. To communicate from an ACP-127 channel using MoRaSky's simulated serial hub, use the `morasky` serial proxy.

Running the **morasky** command without arguments shows a helptext which describes the valid command-line options. A common configuration is to run 3 simulated modems on localhost, which is done as follows:

```
morasky -P 3 127.0.0.1
```

This sets up modems on ports 58001, 58002 and 58003, and the serial hub on port 58000. If connections need to be made from other machines or VMs, then use 0.0.0.0 to cause MoRaSky to bind to all interfaces:

```
morasky -P 3 0.0.0.0
```

Stop MoRaSky using Ctrl-C. Options to control the simulation (as below) should be added after `morasky` and before `-P`.

## 6.4 Interference simulation

This simulates periods of interference that start and end abruptly. Where the transition occurs within an interleaver block, the FEC coding ratio of the waveform is used to judge whether the data is likely to be recoverable or not. If the block is definitely unrecoverable then it is XOR'd with random data to simulate corruption.

For waveforms with a continuous interleaver, the span of data that would definitely be corrupted by the given period of interference is simulated, and random data is XOR'd with that span.

Naturally nothing is quite so clear-cut in real life, but this simulation aims to give a reasonable approximation of the interaction of abrupt interference with the different waveforms.

Two forms of interference are available: regular (`-ir` *`interference-ms/clear-ms`*) and Markov-chain (`-im` *`interference-ms/clear-ms`*). The times are specified in milliseconds.

Regular interference simply alternates between a fixed period of interference and a fixed period of clear signal. So for example `-ir 1000/9000` would give interference for 1s then clear for 9s, then interference for 1s, clear for 9s, and so on.

For Markov-chain interference, the times are instead used as half-lives for the "interference" and "clear" states. One state changes at random to the other with a constant probability per unit time determined from the state's half-life. This means that the interference fluctuates on and off randomly, but with a behaviour that is statistically well-defined over a long period.

## 6.5 Bit error simulation

Bit errors may be added with the following option: `-e` *`ber/cer/csiz/initial-drop/initial-pad/block-dropout-percent`*. Any parameters in that list that are not required can be omitted, and trailing slashes may also be omitted. So in the simplest case, `-e 3` would specify plain 0.1% bit errors. The various parameters are explained below:

**Table 6.1. Bit error parameters**

| | |
|---|---|
| *`ber`* | Specifies BER (bit error rate) in $-\log_{10}$ form. So for example 3 means a BER of 1e-3, and 3.7 means a BER of 2e-4. By default these bit errors are random but evenly distributed in time. |
| *`cer`* | Specifies the CER (cluster error rate) in $-\log_{10}$ form. This causes the errors specified by the BER value to be clustered into small random groups at the given rate. The $-\log_{10}$ CER should be greater than the BER, e.g. `-e 3/4/5` specifies 1e-3 overall BER, with clusters of errors at a rate of 1e-4. This means that on average there are 10 bit errors per cluster. These bit errors will be randomly spread across a cluster size of 5 bytes. This is used to simulate the clustering of bit errors that occurs due to the nature of the waveform FEC decoding process. |
| *`csiz`* | Cluster size in bytes. Each cluster of bit errors affects this many bytes, i.e. the bit errors belonging to this cluster will be spread randomly across these bytes. |
| *`initial-drop`* | Specifies the number of bytes to drop at the beginning of a transmission. |
| *`initial-pad`* | Specifies the number of bytes of random data to add at the beginning of a transmission. |
| *`block-dropout-percent`* | Specifies the percentage of interleaver blocks that should be dropped entirely from the transmission on reception. This simulates modem behaviour when channel conditions get very bad. Normally the modem will continue to output data in bad conditions, even as the BER approaches 50%. The parameters above simulate that case. However, beyond a certain point the modem starts to lose whole blocks, and this parameter can be used to simulate that case. |

All of the behaviours above have been observed in real modems running long-running tests over channel simulators. They can be used to test that software remains resilient even in the case of unexpected conditions.

## 6.6 SNR-based bit error generation

The fixed error conditions generated by the `-e` option above have limited value when running longer tests, because real skywave channels do not behave that way. So MoRaSky provides table-driven SNR-based bit error simulation.

The tables were built up by sending known test data between two modems over a channel simulator and then analysing to determine BER, CER, cluster size, and padded/dropped bytes. There are four tables corresponding to different standard channel conditions: AWGN, CCIR-G, CCIR-M and CCIR-P (for additive white gaussian noise, CCIR good, CCIR medium and CCIR poor). The tables are indexed by waveform type, bit rate, interleaver size and SNR.

This means that it is possible to simulate varying SNR with a fixed waveform, or varying the waveform with fixed SNR, or both. The aim is to make a long-running simulation contain the unusual conditions that might occur in real life. So depending on simulated channel conditions, any of the configurable behaviours for the `-e` option above may be triggered, i.e. anything from perfect transmission to bit error clusters to continuous saturated bit errors, perhaps with dropped/padded bytes or even whole dropped blocks.

The `-sf` *`snr-db`* option selects a fixed SNR value. For example:

```
morasky -t CCIR-G -sf 20 -P 3 127.0.0.1
```

Here the CCIR-G table and a fixed SNR of 20dB are selected. The error conditions will be selected for each transmission from the table according to the current waveform.

The `-sw` *`snr`*`{,`*`snr`*`...}` option selects Walnut Street model SNR generation. This is intended to simulate conditions that might be found in a skywave channel. For example:

```
morasky -t CCIR-G -sw 5,10,15,20 -P 3 127.0.0.1
```

The SNR values listed represent the long-term SNR trend. The baseline is formed by placing the provided SNR values at 1 hour intervals and linearly interpolating the SNR values (in dB) between them. The final SNR value is then held constant. (If only one value is provided, it forms a constant baseline SNR value.)

The Walnut Street variation is applied on top of the baseline. This consists of lognormal SNR variations with standard deviation of 4dB and time-constant of 10s. This means that the SNR (and therefore the generated errors) may change several times within a single transmission.

The `-sv` option specifies variable SNR generation. This allows both the table and SNR to be controlled remotely over the RAP1 protocol using 201/9999 messages. **hftool** can use this to treat MoRaSky as a virtual channel simulator.

The `-sd` option is used to dump out the SNR tables that MoRaSky uses internally to drive its SNR-based bit error simulation. It requires both `-wf` and `-t` options to select the part of the table to dump. If the `-wf` option is incomplete or missing, the list of possible `-wf` values will be shown. Otherwise the relevant table entry is dumped out. For example:

```
morasky -wf 'wf=4539 bps=300 ilv=S' -t CCIR-G -sd
```

gives:

```
Table: 4539 300 S, unsmooth
  drop zero: 0.0, gradient 0.0
  pad zero: 0.0, gradient 0.0
  Entry SNR -4.0, BER 0.51, CER 2.65, Csiz 35.7
  Entry SNR -3.0, BER 0.71, CER 2.72, Csiz 26.0
  Entry SNR -2.0, BER 0.95, CER 2.73, Csiz 16.3
  Entry SNR -1.0, BER 0.98, CER 2.77, Csiz 16.7
  Entry SNR 0.0, BER 1.26, CER 3.39, Csiz 34.1
  Entry SNR 1.0, BER 0.99, CER 3.19, Csiz 40.8
  Entry SNR 2.0, BER 1.7, CER 3.33, Csiz 12.6
  Entry SNR 3.0, BER 1.4, CER 3.23, Csiz 18.2
  Entry SNR 4.0, BER 1.94, CER 3.76, Csiz 16.3
  Entry SNR 5.0, BER 2.24, CER 3.53, Csiz 7.0
  Entry SNR 6.0, BER 1.71, CER 3.19, Csiz 8.5
  Entry SNR 7.0, no errors
  Entry SNR 8.0, no errors
  Entry SNR 9.0, no errors
  ...
```

This shows the entry in the "CCIR Good" table corresponding to the 4539 waveform for 300 bps and a short interleaver. Drops and pads (as irregular events) are handled with a line representing a probability plotted against SNR, which in this case is flat and zero. The simulation will be based around the BER/CER/Csiz values listed for each SNR value, with a small amount of SNR randomisation to reflect real life variation.

---

**Note:** The aim of the SNR-based simulation is to provide enough semi-realistic variation to properly exercise software attempting to transmit data over the simulated channel. The simulation might have properties that differ from a genuine skywave channel. If different simulated channel properties are required for a particular test, then Isode would consider extending MoRaSky with a new simulation to provide them.

---

## 6.7 Crypto box simulation

Crypto boxes are normally deployed on the data stream passing into and out of the modem. Data is encrypted on sending and decrypted on receiving. For the purposes of MoRaSky we are not concerned about the security provided by the cryptography, but rather about the effect that it has on the data stream as seen by the software at the receiving end. Bit errors and interference have a different effect on an encrypted transmission compared to an unencrypted one. This is what we aim to simulate here.

Three forms of crypto box simulation are provided: `-c1` for BID1650-style crypto, `-c2` for Crypto AG with short headers and `-c3` for Crypto AG with long headers. For example:

```
morasky -c1 -P 3 127.0.0.1
```

The simulations scramble the data with very simple (non-secure) algorithms using a key value held in the header, and then unscramble the data again at the receiving end. The scrambling algorithms are intended to give a similar response to bit errors as real crypto boxes. For example, a bit error in a header is likely to corrupt the entire transmission, whereas a bit error in the body of the data will have a more limited and localised effect. Where longer headers are available, it is supposed that this gives some redundancy, meaning that bit errors in the header are less likely to corrupt the whole stream.

Some crypto boxes overwrite the initial part of a transmission with their header. The BID1650 boxes do this when a sync serial interface is used. As the `-c1` crypto simulation cannot detect what kind of serial interface is used, it always overwrites the initial 13 bytes of the transmission. This means that the sending software must be configured to add 13 bytes of padding at the start of each transmission.

Other crypto boxes always insert a header before the data. This affects software that may be attempting to tune the transmission length to use a whole number of interleaver blocks, which must then be configured to take account of the extra space used by the crypto header. Since the simulation also inserts the same number of header bytes, this will affect the transmission in the same way by causing an unexpected extra interleaver block to be transmitted. This configuration error should be visible if the overall average throughput is compared between long runs with and without the crypto simulation enabled.

## 6.8 Miscellaneous options

**Table 6.2. Miscellaneous options**

| | |
|---|---|
| `-L` | Generate output suitable for a log-file rather than a terminal. Normally MoRaSky shows a status line on the bottom line of the terminal, but this is unhelpful if the output is going to a log file. This option disables the status line output. |
| `-W` | Wait on exit. This can be used on Windows to keep the terminal window open and to allow the output to be viewed. |
| `-wf "`*`wf-spec`*`"` | Specify the initial waveform to be configured in the modems. Normally **hftool** or Icon-5066 configures the waveform, but if MoRaSky is being accessed over the serial hub, then this option is required to specify the waveform to use. See Waveform Specifications below. |
| `-bp` *`port-number`* | Specify the base port number to use instead of 58001. For example if 40001 is specified, then modems will listen on 40001, 40002, etc, and the serial hub will listen on 40000. |
| `-r` | Specify reproducible transmission errors. MoRaSky uses randomness to make the simulation more realistic. However, for automated testing it is helpful to be able to repeat the same test over and over, with exactly the same conditions. With `-r` specified, the random number sources generate the same sequences each run. If the same sequence of transmissions is made, then the same bit errors and interference will be introduced each time. |
| `-V` *`port-number`* | Run in virtual time coordinated by a lockstep server running on localhost with the given port. This is used for Isode-internal accelerated testing. |
| `-mr` or `-mp` | Use RM6-like modem timings (`-mr`, the default) or 'perfect' modem timings (`-mp`). The perfect timings only cover what is required to generate the waveform. The RM6-like timings are more realistic taking into account processing delays and timeouts internal to the modem. |
| `-B` | Bypass the modem/radio simulation and pass transmitted data directly to the outputs of the other modems. |

## 6.9 Waveform specifications

A waveform specification takes the form of a string containing a number of *key=value* pairs separated by spaces, for example: `wf=4539 bps=300 ilv=S`. The following table shows the allowable combinations. Any combination of elements from the same table row is permitted, except for the case of `wf=5069` where the allowable combinations depend on the bandwidth setting; see MIL-STD-188-110C for details.

**Table 6.3. Valid waveform specs**

| | | | |
|---|---|---|---|
| wf=4285 | | bps=75 bps=150 bps=300 bps=600 bps=1200 bps=2400 | ilv=S<br>ilv=L |
| wf=4285 | | bps=1200 bps=2400 bps=3600 | ilv=Z |
| wf=4539 | | bps=75 bps=150 bps=300 bps=600 bps=1200 bps=2400 | ilv=S<br>ilv=L<br>ilv=Z |
| wf=4539 | | bps=3200 bps=4800 bps=6400 bps=8000 bps=9600 | ilv=US<br>ilv=VS<br>ilv=S<br>ilv=M<br>ilv=L<br>ilv=VL |
| wf=5069 | bw=3<br>bw=6<br>bw=9<br>bw=12<br>bw=15<br>bw=18<br>bw=21<br>bw=24 | bps=75 bps=150 bps=300 bps=600 bps=1200 bps=1600 bps=2400 bps=3200 bps=4800 bps=6400 bps=8000 bps=9600 bps=12000 bps=12800 bps=14400 bps=16000 bps=19200 bps=24000 bps=25600 bps=28800 bps=32000 bps=36000 bps=38400 bps=40000 bps=48000 bps=51200 bps=57600 bps=64000 bps=72000 bps=76800 bps=90000 bps=96000 bps=115200 bps=120000 | ilv=US<br>ilv=S<br>ilv=M<br>ilv=L |

## 6.10 Serial hub configuration

When running tests over the serial hub simulation, there is nothing driving the modems directly, so it is important to remember to add the `-wf` option to select the modem waveform. For example:

```
morasky -wf 'wf=4539 bps=9600 ilv=S' -P 2 0.0.0.0
```

The simulated serial hub uses the Isode serial proxy protocol over TCP/IP, by default listening on port 58000. This allows realistic simulation of flow control for both sync and async serial. This is normally driven directly from the `morasky` serial proxy. The configuration is expressed in two strings of space-separated *key=value* pairs, one for serial port selection, and the other for serial port configuration. These must be passed to the serial proxy to configure the serial port. It is not possible to configure the serial ports from the MoRaSky command-line.

The serial port selection parameters are as follows:

**Table 6.4. Keys for serial OPEN**

| | |
|---|---|
| host=*ip-address* | Selects the MoRaSky IP address to connect to, defaulting to 127.0.0.1 if not specified. |
| port=*port-number* | Selects the MoRaSky port to connect to, defaulting to 58000 if not specified. |
| dev=*serial-port* | Selects the serial hub port to use: `dev=0` addresses the serial port connected to the first simulated radio, `dev=1` the second radio and so on. |

The serial port configuration parameters are as follows:

**Table 6.5. Keys for serial CONFIG**

| | |
|---|---|
| sync=*0-or-1* | Selects asynchronous serial mode (0, the default), or synchronous serial mode (1) |
| baud=*rate* | Selects baud rate of transmission for async serial. Whilst normal serial ports are limited to certain fixed baud rates (75, 150, 300, etc), the simulation can accept any value here. For sync serial, the simulated serial interface is clocked by the modem at the waveform data rate, so this parameter is not required in that case. |
| data=*5-6-7-or-8* | Specifies the number of data bits per byte, from 5 to 8. Default is 8 if not specified. |
| parity=*N-E-or-O* | Specifies the parity for async serial: `N` for none, `E` for even or `O` for odd. Default is none if not specified. |
| stop=*1-or-2* | Specifies the number of stop bits for async serial: 1 or 2. Default is 1 if not specified. |

# Chapter 7 Modem and Driver Testing

This chapter explains how to test modems and their drivers using the Icon-5066 **hftool** command.

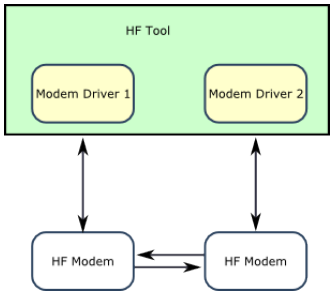
## 7.1 Target Audience

HF Tool provides a flexible way to exercise a modem and modem driver to ensure that a wide range of usage patterns are covered. It was designed to exercise a modem driver, but as a consequence also provides detailed testing and measurements of the underlying modems and any connecting radio systems. HF Tool is intended for the following classes of users:

- Developers of modem drivers for Icon 5066.
- For modem vendors and suppliers, to verify that modem drivers work correctly with different modems using the same interface and with different modem firmware/software versions.
- For solution providers and systems integrators to enable testing of the modem sub-layer independently from the applications, including modem drivers, modems and radios.

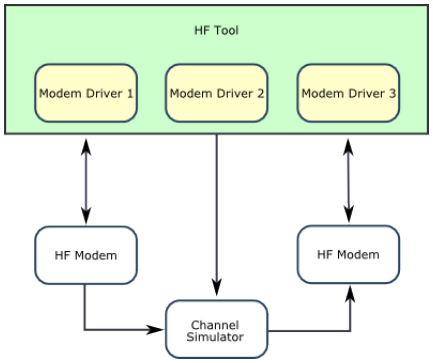
## 7.2 Testing Configurations

**Figure 7.1. hftool controlling two modems**



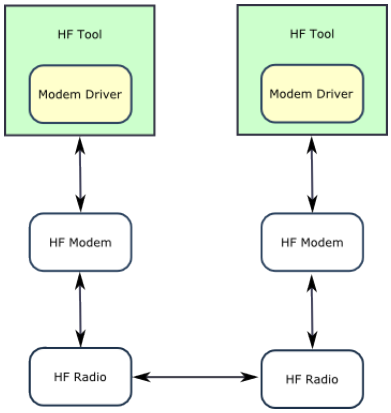

The modems are connected directly to each other. The **hftool** controls both modems and gathers information from both sides of the transmission process. With a forwards connection only, it is possible to run **err** error-rate, **blk** block and **len** length tests. With the addition of the reverse connection, it is possible to run **turn** turnaround tests.

**Figure 7.2. hftool controlling two modems and a channel simulator**




The modems are connected via a channel simulator. In this configuration the **err** error-rate test is able to scan the channel simulator SNR ranges to test error performance.

**Figure 7.3. Two remote modems controlled by separate hftool instances**




Two **hftool** instances control two separate modems, perhaps at remote locations and connected over-the-air via radios. The **ota** over-the-air test sends a sequence of transmissions of known test data with different waveforms from one side, whilst another **hftool** instance records and analyses the received data on the other side. Both channel quality information (such as SNR) and bit error analysis is recorded.

## 7.3 HF Tool Usage Examples

The **hftool** command is provided for testing the modem drivers independently from the Icon 5066 server. It interfaces directly to the modem driver back-end of the Icon-5066 server and uses the same Lua modem drivers that the server uses, through the same Lua API. It can simultaneously control up to one transmit modem, one receive modem and one channel simulator.

It is designed for testing the correct operation of:

- **Interface:** The Lua API and its support code in C
- **Drivers:** The modem drivers written in Lua
- **Modems:** The modems: transmitting and receiving in all their waveform/bps/interleaver combinations
- **Error-Handling:** The modem and driver when running over a channel with interference

The **hftool** is located under *(SBINDIR)* on both UNIX and Windows. Below are some example command-lines to show how to use **hftool**.

### 7.3.1 Display usage

Running the command with the **-?** argument gives a usage message:

```
hftool -?
```

This describes a full list all of the current options, some of which might not be described in these examples.

### 7.3.2 List waveforms

List all supported waveforms for the given driver (all combinations of waveform, bps-rate and interleaver):

```
hftool -d rockwell_collins ls
```

Filter that list to just 4285 waveforms:

```
hftool -d rockwell_collins ls wf=4285
```

Filter that list to 4285 and 4539 waveforms with a short interleaver (comma is used for lists of values):

```
hftool -d rockwell_collins ls wf=4285,4539 ilv=S
```

Filter that list to all waveforms with rates in the range 1000 to 3000 (range separator is minus or colon):

```
hftool -d rockwell_collins ls bps=1000-3000
```

Filter that list to all 5069 waveforms with bandwidth of 3 or 12 to 24 and a long interleaver:

```
hftool -d rockwell_collins ls wf=5069 bw=3,12-24 ilv=L
```

The option **-d** selects the modem driver to use. It is the name of the Lua file that implements the driver, without the trailing *.lua*. The driver is searched for first in *(ETCDIR)/s5066/drivers/* then *(SHAREDIR)/s5066/drivers/*.

These same filters are used in all the commands below to select the list of waveforms that the test should run with. All terms in the filter are combined using an AND operation, i.e. all must be matched for a waveform to be listed or used. For more complicated filters, nested AND and OR operations are available using an advanced syntax described in @@@ below.

### 7.3.3 Basic testing

For connecting modems for testing, use **-R** for a receive modem and **-T** for a transmit modem. These use the **-d**, **host=**, **port=**, **type=** etc settings preceding them. **-d** selects the modem driver. If required by the driver, **host=** gives the IP address of the modem, **port=** gives the port number and **type=** gives the modem type. Not all drivers need all of these specifications.

For example, this command sets up a receive modem on 192.168.1.221 and uses **rx** to listen indefinitely for data transmitted with the 4539 waveform:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  rx wf=4539
```

This sets up a transmit modem on 192.168.1.222 and sends 10 seconds of test data on each of the matched waveforms:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.222 -T \
  err 10 wf=4539 bps=1200
```

This does both at the same time, and displays the received data in full (**-v**):

```
hftool -v -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  err 10 wf=4539 bps=1200
```

### 7.3.4 Channel simulators

Channel simulators can also be driven, where a driver has been written for them. A channel simulator may have its own dedicated driver, but if the same device functions as both a modem and a channel simulator, then the same driver may be used to cover both. The **-C** option configures a channel simulator, similar to how **-R** and **-T** configure modems. The first argument is the simulator preset to use (which depends on what the driver supports), and second argument is the SNR value in dB to use, or a range of values separated by minus or colon. For example:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T
  host=192.168.1.218 -C CCIR_P 10 \
  err 10 wf=4539 bps=1200
```

Most tests don't need to change the SNR, so the highest SNR given is applied to the simulator. For tests that change the SNR (e.g. the error-rate test), the entire range is used.

Note that it is possible to drive each of the two modems and the channel simulator with different driver (**-d**) options, for cross-modem testing. As an illustration, using a fictional 'dsp-box' driver:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  -d rap1 host=192.168.1.222 -T \
  -d dsp-box host=192.168.1.218 -C CCIR_P 10 \
  err 10 wf=4539 bps=1200
```

The channel simulator is most useful in combination with the **err** error-rate test below.

### 7.3.5 Block timing test

This tests that the modem interleaver block lengths are as expected by the driver, that there are no problems making small transmissions, and that transmission overheads are as expected. Icon-5066 uses knowledge of the waveform interleaver block sizes to precisely fill the final block of each transmission, so these calculations must be correct.

Just less than one block, two blocks and three blocks are transmitted on each of the waveforms. This allows the driver-writer to check that there are no unexpected delays in the handling. The reason for sending just less than one block is to test that the driver is

flushing the modem correctly and that the modem is not timing out (for modems that behave that way). The example below runs the test for all 4539 and 5069 waveforms:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  blk wf=4539,5069
```

Here is some example output from the **-o** option, or after applying **grep** '^>' to the general output:

**Figure 7.4. Example output from hftool blk**

```
>#WForm BW Rate   IL       TX-startup-------  TX-time-span-----  RX-time-span-----  Data-in-time-span
>#                    Unit    x1    x2    x3    x1    x2    x3    x1    x2    x3    x1    x2    x3
> 4539    75    S      6     6     6     6,  3627  4230  4829,  2275  2945  3544,  1201  1801  2401
> 4539    75    L     23     3     6     6,  9627 14430 14429,  4758  9436  9563,     0  4800  4801
> 4539   150    S     11     6     6     5,  2430  3031  3628,  1699  2350  2947,  1201  1800  2402
> 4539   150    L     68     3     6     6,  9629 14426 19229,  4769  9484 14285,     0  4802  9600
> 4539   300    S     23     3     6     6,  1829  2430  3030,  1103  1748  2350,   599  1200  1801
> 4539   300    L    158     6     7     7,  9627 14429 19230,  4757  9490 14291,     0  4801  9602
> 4539   600    S     23     6     6     6,  1227  1830  1830,   499  1152  1150,     0   601   602
> 4539   600    L    338     6     7     7,  9629 14429 19230,  4759  9488 14291,     0  4800  9599
> 4539  1200    S     68     3     6     6,  1230  1829  2427,   511  1150  1748,     0   601  1199
> 4539  1200    L    698     6     7     8,  9630 14430 19231,  4766  9496 14304,     0  4801  9602
> 4539  2400    S    158     6     6     7,  1231  1830  2431,   523  1152  1752,     0   601  1201
> 4539  2400    L   1418     8    10    11,  9629 14427 19226,  9035  9505 14314,  7891  4801  9602
```

**Table 7.1. Columns in hftool blk output**

| | |
|---|---|
| WForm | Waveform (4285, 4539, 5069, etc) |
| BW | Bandwidth in kHz |
| Rate | BPS rate |
| IL | Interleaver |
| Unit | Block-length unit in bytes used for the test. (This might not be a real block length for waveforms that don't have blocking.) |
| TX startup | Time between requesting a transmission and the modem reporting that the transmission has started, for transmission of one, two and three blocks, in ms. |
| TX over expected | How much the transmission time exceeds the expected time (calculated by the modem driver), for transmission of one, two and three blocks, in ms. |
| TX time-span | Time between transmitting modem reporting transmission start and stop, for transmission of one, two and three blocks, in ms. |
| RX time-span | Time between receiving modem reporting reception start and stop, for transmission of one, two and three blocks, in ms. |
| Data-in time-span | Time between first data received from modem and last data received, in ms. |

#### 7.3.5.1 Interpreting the results

Check:

- "TX startup" and "TX over expected" should be roughly the same for 1, 2 and 3 blocks on each line, and consistent with other waveforms.
- "TX time-span", "RX time-span" and "Data-in time-span" should increase in even steps from 1 block up to 3 blocks. Each step should be about the length of the interleaver block in ms.

Where the results don't fit this pattern, it's necessary to consider what might be wrong:

- The block length calculations in the driver may differ from the modem's calculations.

- The modem might be waiting to time-out before sending instead of sending right away, e.g. if there isn't a command to indicate that the outgoing data has finished and that the modem should flush and start the transmission.
- There may be some other problem with the streaming or buffering on the way to the modem causing a delay.
- Data may be being inserted and deleted by something else in the data path to the modem (e.g. crypto device), and the drivers haven't been correctly configured to take account of that.

Any problems with block-length calculations should be fixed before deploying because otherwise there will be wasted space at the end of all the transmissions.

### 7.3.6 Length timing test

The modem driver can be asked to calculate an estimate of how much data can be transmitted in any given time. This is supported by standard tables (*waveform.lua*), whose output the modem driver may adjust to tailor for the specific modem's behaviour.

The length test tests this length estimation against the actual modem behaviour. Various different lengths of transmission are selected, and the driver is asked to calculate how many bytes can be transmitted in that time. The transmission is made and the actual transmission time is compared to the estimate.

This will show up estimation errors, unexpected delays in the modem handling, and also driver coding errors such as configuring the wrong bps rate. The argument to the **len** option is the starting number of seconds for the transmission. For example, testing only 5069 waveforms with bandwidth of 24kHz:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  len 20 wf=5069 bw=24
```

**Figure 7.5. Example output from hftool len**

```
>#WForm BW Rate   IL   Target Expect Actual Under   Over
> 4285     600    S     60000  59946  59869
>  Transmission 0: pad 0, brk 0, 1.5% loss, 0.29 -log10 BER
> 4285     600    S     30000  29973  29899
> 4285     600    S     15000  14933  14856
> 4285     600    S      7500   7466   7393
> 4285     600    S      3750   3733   3655   97%
> 4285     600    S      1875   1813   1739   95%         -
> 4285     1200   S     60000  59946  59983          37ms  #
>  Transmission 0: pad 0, brk 0, 1.4% loss, 0.18 -log10 BER
> 4285     1200   S     30000  29973  30006          33ms  #
> 4285     1200   S     15000  14933  14963          30ms  #
> 4285     1200   S      7500   7466   7500          34ms  #
> 4285     1200   S      3750   3733   3763          30ms  #
> 4285     1200   S      1875   1813   1842          29ms  #
```

If a receive modem is configured, the test also checks that the data is received correctly. In the report above, two transmissions were not received correctly, probably due to the receiving modem not being quite ready after the change of waveform.

**Table 7.2. Columns in hftool len output**

| | |
|---|---|
| WForm | Waveform (4285, 4539, 5069, etc) |
| BW | Bandwidth in kHz |
| Rate | BPS rate |
| IL | Interleaver |
| Target | Time we are aiming to transmit for, in ms |
| Expect | Expected transmission time, as reported by driver (calculated from waveform tables), in ms. |
| Actual | Measured transmission time (between transmission start and end) as reported by the transmitting modem. |

| | |
|---|---|
| Under | If the actual transmission time is more than 1-2% under the expected time, this gives the actual time as a percentage of the expected time. |
| Over | If the actual transmission time is over the expected time, then the excess is reported here, in ms. |
| Bar chart | Variations from the expected time are indicated here using one or more minus signs for under, or hash signs for over. This helps large errors stand out in the report. |

#### 7.3.6.1 Interpreting the results

Check:

- "Over" and "Under" values should be small and relatively consistent between different lines in the report.

Anything that stands out needs explaining. Some possible causes of problems:

- Incorrect block-length calculations by the driver, noticeable especially for larger block sizes, depending on how the lengths align.
- Incorrect configuration of the waveform in the modem (e.g. wrong BPS or interleaver).

### 7.3.7 Ping-pong turnaround test

This is to test how quickly the modems can swap roles, both when changing waveform and when continuing with the same waveform. A fixed length of data is transmitted one way, and then the modems swap roles and the same data is transmitted back the other way. This is repeated again without changing the waveforms, to make 4 transmissions (2 round trips) per waveform.

There are two parameters: the duration in seconds of the data to send each way, and the minimum turnaround time to enforce in ms. Start with a minimum turnaround time of 0 to get accurate measurements. If the modems aren't ready early enough it is possible to increase it until a safe margin is found to avoid errors.

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  turn 20 0
```

**Figure 7.6. Example output from hftool turn**

```
>#WForm BW Rate   IL ------- Turnaround -------- --- Mean ---- ------- Transmission ------
>#                    WF changed  WF unchanged WFconf/TX-RX
> 4539     8000   US    520    340    326    229    156/275     19939  19939  19939  19939
> 4539     8000   VS    508    360    358    192    141/284     19883  19903  19903  19902
> 4539     8000   S     474    434    375    196    143/298     19580  19581  19579  19581
> 4539     8000   M     513    470    411    303    144/352     19541  19541  19542  19541
> 4539     8000   L     631    513    505    407    140/444     17360  17358  17364  17363
> 4539     8000   VL    906    788    712    592    141/679     17397  17398  17400  17400
> 4539     9600   US    479    369    356    190    140/279     19903  19903  19903  19902
> 4539     9600   VS    540    314    339    254    138/293     19904  19904  19903  19903
> 4539     9600   S     500    371    354    256    141/300     19580  19582  19581  19581
> 4539     9600   M     652    417    399    276    141/365     19581  19582  19542  19581
> 4539     9600   L     657    573    519    412    141/470     17400  17400  17359  17360
> 4539     9600   VL    908    795    717    668    144/700     17401  17397  17399  17399
```

**Table 7.3. Columns in hftool -L output**

| | |
|---|---|
| WForm | Waveform (4285, 4539, 5069, etc) |
| BW | Bandwidth in kHz |
| Rate | BPS rate |
| IL | Interleaver |
| Turnaround | The turnaround time is made up of two parts: the time from starting the transmission to the transmitting modem reporting the transmission has started, and the time from the transmitting modem |

| | |
|---|---|
| | reporting that the transmission has finished, to the receiving modem reporting that reception has finished and all data having arrived. When the waveform is also changed (first two columns) the time to change waveforms is also added to the figure. |
| Mean WFconf | This is the average time for the waveform configuration, in ms, calculated from the four turnaround figures. |
| Mean TX-RX | This is the average time required to turnaround the waveform attributable to RX and TX actions (e.g. starting off transmission, tail-end of waveform decoding, flushing buffers, etc), in ms, calculated from the four turnaround figures. |
| Transmission | These are the transmission times for each of the transmissions as measured between transmission start and end reported by the transmitting modem, in ms. |
| Mean Over | This is the average amount that the transmission times are over the expected transmission time. This is the same value measured in a different way in the length test. It is shown here to allow the total wasted time to be assessed. |

#### 7.3.7.1 Interpreting the results

Check:

- Check down the "Mean WFconf/TX-RX" figures and check that they are consistent with one another from line to line, and seem realistic. Variation with interleaver length may be explained by the extra signal processing time required for longer interleavers at the end of a transmission.
- Check down the "Mean Over" column to see that it appears consistent and realistic. Look for anything unusual.

Anything that stands out needs investigating. Really the aim is to have as fast a turnaround as possible. Causes for slow turnaround might include:

- Long interleavers that require a lot of modem signal processing time at the end of a transmission.
- Slow data communication to/from the modem. It is not possible to do a turnaround in STANAG 5066 until all data has been received (or else ARQ D_PDUs would have to be retransmitted), and it's not possible for the modem to start transmitting until its buffers have been prefilled to the required level. So slow communication will automatically add time to both halves of the turnaround.
- For short transmissions (below the prefill level) an extra delay may occur in cases where the modem doesn't have a command to tell the modem to flush and commit to a transmission.

### 7.3.8 Streamer testing

The streamer component is responsible for getting the data to the modem in good time to avoid underflow, but as late as possible to allow urgent data to queue-jump and get transmitted without too much delay. It allows a certain leeway in case of scheduling delays on the server. The streamer test allows testing those limits using artificial delays. Up to near the limit, things should operate normally. Beyond that point, streaming should break down and the modem will likely underflow and break the transmission into two parts. This should be detected and reported.

The **str** command invokes the streamer test. The first argument is the transmission length in seconds. The second argument selects the artificial delay to add to test the streaming, in seconds (which may be fractional). Alternatively if the second argument is -1, then a delay is selected that will exceed the limit by 100ms, which should trigger an underflow.

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  str 60 1.9 wf=4539 bps=9600 ilv=L
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  str 60 -1 wf=4539 bps=9600 ilv=L
```

#### 7.3.8.1 Interpreting the results

Check:

- If the delay is shorter than the allowed time (i.e. usually below 2s), expect perfect transmission of data, with no flapping (multiple transmissions) or missing data or underflows reported.
- If the delay is longer than the allowed time (e.g. specifying -1), expect a report of underflow, and possibly other warnings or errors.

If streaming is not operating as expected, check for causes:

- For the case of not expecting an underflow but one occurring, check that the machine is not busy with other jobs, e.g. virus disk scan, background tasks, etc. We are intentionally pushing things to the limit to test them, so extra load on the machine may push timings over the edge.
- The streamer may be misconfigured, or the wrong type of streamer might be being used. Different modem or serial drivers support the three streaming models differently.

### 7.3.9 Error-rate testing

This involves the **hftool** controlling both transmit and receive modems, sending a known test-pattern and checking the received data for bit errors, dropped bytes and padding. It repeats the test over each of the waveforms matched.

The **err** command invokes the error-rate test. Its argument is the number of seconds of data to send for each wf/bps/ilv combination. For example, testing with 10 seconds of data for each of the 4539 waveforms:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  err 10 wf=4539
```

For testing two remote modems with separate **hftool** invocations, one command specifies the receive modem, the other the transmit modem. The receive side should be started before the transmit side. Both ends must use the same **err** parameter because the receive side uses this to know how much data it should expect. Analysis is generated on the receive side, but any timing measurements that depend on times from the transmit side will be absent from the report:

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  err 10 wf=4539
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.222 -T \
  err 10 wf=4539
```

If a channel simulator is set up, the error-rate test will first do one transmission in calibration mode, then do a scan through the range of SNR values provided, making a transmission at each one, to build up a table of measurements to characterise the behaviour of the waveform and modem with that channel simulator configuration. Not all SNR values are tried: a search is performed to make the test quicker and to focus on the region of change between perfect reception and no reception (allowing a bit of leeway).

```
hftool -d rockwell_collins type=HSM-2050 host=192.168.1.221 -R \
  host=192.168.1.222 -T \
  host=192.168.1.218 -C AWGN -10:40 \
  err 10 wf=4539
```

The above command will build up a table of results for each of the waveforms matched by the filter. In this case, the SNR range scanned is from -10 dB to 40 dB, according to the specification "-10:40", using the AWGN channel-simulator preset.

Summary lines in the output are preceded by `'>'` so can be extracted from the general output with **grep** `'^>'` on UNIX. Alternatively, use the `-o` *`filename`* option to output summary lines to a separate file. For example:

**Figure 7.7. Example output from hftool err**

```
>#Chan-Sim-                    ---------- Measured ----------  -Reported-              Time-span of:
>#IF    SNR WForm BW Rate   IL    Pad Brk Loss%  BER  CER Csiz  SNR  BER  Conf Start Detct    TX    RX  Data
>  AWGN   6 4285     2400   L   8571   -  58.3 0.29    -    -  5.9    -     -     6   256 59975 59554 49175
>  AWGN   7 4285     2400   L      0   -   0.7 0.27    -    -  7.1    -     -     2   268 59977 59542 49174
>  AWGN   8 4285     2400   L      6   -     - 1.91 2.95  3.5  8.5    -     -     5   337 59977 59472 49495
>  AWGN   9 4285     2400   L      6   -     - 2.60 3.58  2.7  9.1    -     -     5   294 59976 59514 49495
>  AWGN  10 4285     2400   L      6   -     - 3.75 4.47  1.8 10.3    -     -     4   282 59981 59531 49494
>  AWGN  11 4285     2400   L      6   -     -    -    -    -    - 10.7    -     -     3   367 59977 59444 49494
>  AWGN  12 4285     2400   L      6   -     -    -    -    -    - 12.1    -     -     3   282 59978 59529 49495
>  AWGN  13 4285     2400   L      6   -     -    -    -    -    - 13.3    -     -     6   278 59977 59533 49495
```

Note that errors in reception may appear in either the `Loss%` column or the `BER` column, depending on whether there appear to be received bytes which correspond to the transmitted bytes or not.

The known data sent is designed to be recognised even in the presence of constant bit errors, and decoding it gives the offset that it originally occurred at within the data as sent, so insertion or deletion of bytes is detected even at large offsets. It does assume that byte-alignment of bits is maintained throughout, however.

Here is a full description of all the columns:

**Table 7.4. Columns in hftool err output**

| | |
|---|---|
| Chan-Sim IF | Channel simulator interference preset |
| Chan-Sim SNR | Channel simulator SNR level (dB) |
| WForm | Waveform (4285, 4539, 5069, etc) |
| BW | Bandwidth in kHz |
| Rate | BPS rate |
| IL | Interleaver |
| Pad | Measured padding in bytes. Padding bytes are bytes received which do not appear to correspond to the sent data in any way. |
| Brk | Number of breaks in between continuous runs of received bytes which appear to correspond to transmitted bytes. A break might be due to padding bytes or dropped bytes. |
| Loss% | Percentage of transmitted data which never arrived, i.e. to which no corresponding bytes in the received data can be identified |
| BER | Bit Error Rate in the received data which has been identified as corresponding to transmitted data. This is expressed as -log10(error-rate), so 0.30 is 50% BER, 1.00 is 10% BER, and 2.00 is 1% BER. |
| CER | Cluster Error Rate, expressed as -log10(error-rate). A run of 4 or more correct bytes separates clusters of errors. Each cluster is counted as one error, giving a cluster-error-rate. |
| Csiz | Average size of error clusters in bytes. |
| Reported SNR | The measured SNR reported by the modem, if available. |

| | |
|---|---|
| Reported BER | The estimated BER reported by the modem, if available. |
| Conf | Time required to configure the waveform, in ms. |
| Start | Time between asking the modem to start the transmission, and the modem reporting that transmission has started, in ms. |
| Detct | Time between the transmitting modem reporting that the transmission has started, and the receiving modem reporting that a signal has been detected. |
| Time-span of TX | Time between transmitting modem reporting start and end of transmission, in ms. |
| Time-span of RX | Time between receiving modem reporting start and end of transmission, in ms. |
| Time-span of Data | Time between first data block received and last data block received, in ms. May be 0 if all the data arrived in one block. |

The timing measurements are intended to allow unexpected delays to be detected, analysed and fixed.

To get a crude summary of SNR levels for a BER of 1%, do **grep '^%'** on the output. This is crude because it merely shows the lowest SNR that gets a BER of 1% or less and is free of other errors, without trying to smooth out or interpolate the random variations in the data.

Adding the **-a** option displays a full analysis of the received data using ANSI escapes to highlight bytes with bit errors in red, and showing locations of dropped and padded bytes.

### 7.3.10 Over-the-air channel test

This is designed for over-the-air testing between two distant locations. One location transmits fixed test data on a set of different waveforms, optionally with the waveforms repeated indefinitely on a cycle. The other location accepts all data that the radio and modem are capable of receiving and records a full analysis of bit errors, drops, skips and any channel quality information reported by the modem (such as SNR). These can be used later for offline analysis or graphing, and as a basis for simulating how different types of traffic would have interacted with the channel conditions found at the time of recording.

On the receiving side, the example command below sets the modem up to listen for 4539 waveforms. The **ota** option's parameter of **10** configures the test to wait 10 seconds maximum for data to finish coming in after the modem declares that carrier has dropped. The information recorded is appended to the file 'ota-rx-log':

```
hftool -t ota-rx-log -d rockwell_collins type=HSM-2050 \
  host=172.16.6.71 -R \
  ota 10 wf=4539
```

On the transmitting side, a range of waveforms of interest needs to be selected. These would normally be ones that it is known are reasonably feasible for the channel to carry. In the test below, 4539 waveforms from 3200bps to 9600bps are tested with L and VL interleavers. The **-r** (repeat) option tells **hftool** to repeat the set of waveforms over and over indefinitely.

The parameters to the **ota** option are the transmission length in seconds (600 seconds in this example) and the gap to leave between transmissions (20 seconds in this example). The gap time must be longer than the wait time configured on the receive end by a sufficient margin, or else the receive end will not be able to distinguish between separate transmissions.

```
hftool -t ota-tx-log -r -d rockwell_collins type=HSM-2050 \
  host=172.16.6.72 -T \
  ota 600 20 wf=4539 bps=3200-9600 ilv=L,VL
```

Note that when testing locally with a channel simulator, the channel simulator setup may be configured with the same **hftool** instance used for transmission. Only the maximum SNR value will be configured -- no attempt is made to scan.

**Figure 7.8. Example output from hftool ota at receive end**

```
**QUALITY 00:28:33.877 bps=3200 ilv=M snr=5
**QUALITY 00:28:34.370 bps=3200 ilv=M snr=5
**QUALITY 00:28:34.869 bps=3200 ilv=M snr=4
**QUALITY 00:28:35.367 bps=3200 ilv=M snr=4
**QUALITY 00:28:35.873 bps=3200 ilv=M snr=6
**QUALITY 00:28:36.371 bps=3200 ilv=M snr=6
**START 239324 (analysis dump of 239324 received bytes)
*  ::
*197820  ----------------------------------------3-223----------4----------
*197890  ------------------524653355265241----------3---------7353536552-----
*197960  ----------------------------------3-------------------------------24
*198030  46----------21-------------------------------------------46343------
*198100  ---------------35-----------------------------------645542433432165346
*198170  52575336655335363------------------12-----33552----------------------
*198240  -27674463353523655552-----23664----------------------------34347-------
*198310  --2643----2357541-----------------------------------35433-----------
*198380  ---433---------------------15361--57653-----------------------------
*198450  ------------------------------------------------------------------364563
*198520  62222----------------252-------------532766453-----------------1665466
*198590  3362-----354-------------------------------------2447464343275335346--
*198660  ----35-----------------------------------------------------------------
*  ::
**END
*SENT-LEN-EST 239324
*SENT-VALID-SPANS 0-197864 197869-197879 197880-197909 197924-197935
*SENT-VALID-SPANS 197955-197995 197996-198028 198032-198043 198045-198089
*SENT-VALID-SPANS 198094-198115 198117-198152 198187-198206 198218-198241
*SENT-VALID-SPANS 198270-198298 198327-198363 198368-198383 198386-198408
*SENT-VALID-SPANS 198420-198514 198525-198541 198544-198557 198566-198583
*SENT-VALID-SPANS 198602-198639 198666-239324
* DPDU size:   25   50   75  100  150  200  300  400  600  800 1000
* %Success:    99   99   99   99   99   99   99   99   98   98   98
```

There are several lines of interest in the output. The `QUALITY` lines shows channel-quality information received from the modem driver. This may contain SNR and other signal measures, and also BPS and interleaver waveform settings if reported by the modem.

Lines from `START` to `END` contain a dump of all the data bytes received in the form of a character for each byte which represents the number of bit errors (`1` to `8`) or `-` for no errors. Lines are prefixed with the (decimal) offset within the received data. Where the analysis detects that sent bytes were dropped, a `DROP` line may appear, showing the number of bytes dropped. Similarly a `SKIP` line indicates that unexpected padding bytes were inserted. A line containing `::` shows a region of received bytes with no bit errors.

The `SENT-LEN-EST` line gives an estimate of the length of the sent data according to information available in this received chunk of data. The `SENT-VALID-SPANS` lines are derived from the bit error dump above, and list all the spans of 10 bytes or more which were received correctly. These are expressed in terms of the offsets in the original data as sent (not the offsets as received). Note that each span is exclusive on the upper bound, i.e. `0-4` means bytes at offsets 0, 1, 2 and 3.

Below this appears a quick analysis of how DPDUs of different total lengths would cope with the conditions observed in this last received block of data. This is useful as a rough check, but will be invalid if the transmission is received in several parts.

When reception is difficult, the modem may repeatedly lose carrier and several separate chunks of data will be observed in the log as they were received. For analysis, all of the spans of data reported as successfully received should be merged. The sent data length estimates may be less than what was actually transmitted if it was not possible for the modem to receive any data at the end of the transmission. In this case it may be necessary to cross-reference with the sending logs to see how much data was actually transmitted.

## 7.3.11 Log files

The **hftool** by default writes all logging to the standard output. This includes error messages due to Lua driver errors, and messages written by the Lua drivers using `logwarn()`, `loginfo()` or `tsdb_*()` calls.

It is possible to redirect logging and TSDB data to log files by starting the Isode DDSD daemon, and then specifying the logging directory using the **-l** option.

If the Lua driver fails (e.g. as a result of calling `error()`), the **hftool** will restart the driver, as will the Icon 5066 server. The number of restarts required is noted when the **hftool** terminates.

Lua modem driver failures and warnings should normally be investigated and fixed.

# Appendix A Supported Devices

This appendix lists the devices which are known to be supported by Isode drivers

**Table A.1. Supported devices**

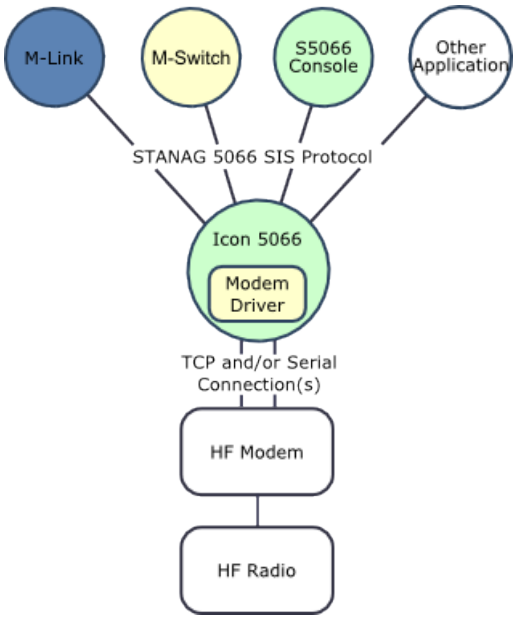
| Driver name | Type | Devices supported | Channel simulator presets |
|---|---|---|---|
| `rockwell_collins` | Modem and channel simulator | Rockwell-Collins HSM-2050, RT-4800, RT-2200A, Q9600 and Q9604. | AWGN RICIAN CCIR_P CCIR_M CCIR_G |
| `rapidm` | Modem | RapidM RM6 and RM8. | |
| `data_only` | Modem | Generic driver for communication with modems over a serial line at fixed speed. | |

# Appendix B Guide for Modem Driver Writers

This chapter explains how to write and test a modem driver for the Icon-5066 server.

## B.1 Introduction

Modem drivers for the Icon-5066 server are written in Lua, which is a lightweight and fast embedded scripting language for C: see http://www.lua.org/. Each modem driver is run in its own private Lua global namespace (i.e. in its own Lua VM). The Icon 5066 server interacts with the driver by calling global functions and callback closures. Lua also supports coroutines which make it easier to write sequential code that works well with the event-driven Icon 5066 server core. It is probably easiest to examine the code of an existing driver to get an idea of how things work.




Isode-supplied drivers and support files are found in *`(SHAREDIR)`/s5066/drivers/* and *`(SHAREDIR)`/s5066/common/*. You may install your own drivers in *`(ETCDIR)`/s5066/drivers/*. Note that *`(ETCDIR)`* is not overwritten on upgrade, unlike *`(SHAREDIR)`*. If you use the same filename in *`(ETCDIR)`* as one of the shipped drivers in *`(SHAREDIR)`*, it will override the shipped driver.

Driver writers may find LuaInspect useful: see http://lua-users.org/wiki/LuaInspect. It is a tool that highlights possible errors in Lua code and it is useful for basic checking of code consistency. It supports a style of Lua coding where `local` variables are used wherever possible. Isode-written drivers use this style.

Note that any fatal error from Lua, caused by invalid code, invalid library calls or calls to the `error()` function results in an error report in the Isode logs and the termination of the driver. Following a failure, the driver will be restarted.

Since the driver code may be called directly from the main loop of the 5066 server, it should never busy-wait or block. Instead it should arrange to be informed via a callback when conditions have changed. If that is not possible, then it should set a timer which rechecks conditions at a later time.

## B.2 Outline Driver Structure

The outline structure of the driver is as follows:

```
require 'modem'
      require 'waveform'
```

These statements pull in functions and definitions required by most modem drivers.

```
function modem_init()
      -- Initial setup, for example:
      host = modem_option_get('host')
      port = modem_option_get('port')
      end
```

This is called to do the initial setup of the driver. It commonly picks up configuration information, such as `host` and `port`. This data is free-form and comes from the XML configuration in the case of Icon-5066, or from the command-line arguments in the case of **hftool**. You could set up the TCP or serial connections to the modem at this point, but that would mean that the driver would need the modem present even just to list waveforms. So it may be better to delay making connections until the first switch-waveform is requested.

For a driver that supports controlling a channel simulator, you can add the code: `cs_spec = modem_option_get("chansim")`. This returns the channel simulator preset name, or `nil` if the driver is not being invoked to control a channel simulator.

```
function modem_close() ... end
```

Called to clean up the driver before driver termination. This may be used to cleanly close connections to the modem, for example. If the driver fails with a Lua error (as created by the `error()` call), then this function will be called before the Lua VM is shut down.

```
function modem_list_waveforms(spec)
      return waveform.list_waveforms({ wf = "4285,4539" }, spec)
      end
```

This should return a list of waveforms supported by the modem, after applying the given filter `spec`. The provided function `waveform.list_waveforms()` does most of the work. The filter passed as an argument to that function should limit the results to the waveforms actually supported by the modem. In complex cases, you'll have to write your own code to filter the list.

Filters are the same as those used for **hftool**, only expressed as a Lua table. `wf` is the waveform name: 4539, 4285 or WBHF (at present), `bps` is the BPS rate, `ilv` is the interleaver, typically: Z, US, VS, S, M, L or VL, and `bw` is the bandwidth of the waveform in kHz, defaulting to 3 if omitted. All values when used in a filter may contain a comma-separated list of valid values. For numeric values a range separated by a minus or colon character may also be used.

The returned value should be a list of tables each of which contains values with the same keys as used by the filter above, but with only a single value for each (i.e. no comma-separated lists or ranges). A truncated example in Lua syntax could be: `{ {`

```
wf="4539", bps="75", ilv="S" }, { wf="4539", bps="150", ilv="S" },
{ wf="4539", bps="300", ilv="S" } }
```

For a channel simulator, this function should return a list of tables containing `snr` values, optionally accompanied by `bw` values. These SNR values indicate the extreme limits of the valid SNR range supported by the simulator (in dB). The `bw` values should be included only if the SNR limits depend on the selected bandwidth. Example without bandwidth dependency: `{ { snr = "-10" }, { snr = "100" } }`, and with bandwidth dependency: `{ { bw = "3", snr = "-10" }, { bw = "3", snr = "100" }, { bw = "6", snr = "0" }, { bw = "6", snr = "100" } }`.

```
function modem_switch_waveform(spec) ... end
```

Start the process of switching the modem to the waveform with the given specification. (The process of switching waveforms is expected to be an asynchronous operation.) The connection to the modem should be initialised now if not active already. Only waveform specifications returned by `modem_list_waveforms()` are expected to work. The code should raise `MODEM_STATE_CONFIGURING` (using `modem_state()`) whilst the modem is being programmed, then drop it and raise `MODEM_STATE_CONFIG_READY` when done. Any failure should be reported with `error()` which will terminate the driver.

For channel-simulator operation, the spec table contains three keys: `calibrate` which is `"1"` (enable) or `"0"` (disable) to control signal level calibration mode, `snr` to configure the dB SNR value, and `bw` to specify the bandwidth to configure for in kHz.

```
function modem_calc_tx_size(wf, seconds)
      return waveform.bytes_for_time(wf, seconds)
      end
```

This function should calculate how much data can be transmitted in the given time for the current waveform. If the modem's timings differ from the calculations performed by `waveform.bytes_for_time()`, then adjustments should be made here to account for the difference.

```
function modem_calc_tx_duration(wf, bytes)
      return waveform.time_for_bytes(wf, bytes)
      end
```

This function should calculate how long it will take to transmit the given amount of data in bytes for the current waveform. If the modem's timings differ from the calculations performed by `waveform.time_for_bytes()`, then adjustments should be made here.

```
function modem_query_tx_params()
      return waveform.tx_params(curr_wf)
      end
```

This function should return three values: the BPS rate, the block size in bytes, and the amount of space to leave free in the final block, all for the currently-configured waveform. The example code above assumes that `curr_wf` contains the current waveform. The 5066 server core will use this information to calculate the maximum transmission length and to pack as much data as possible into the final block. The third return value is used to leave space in the final block for the EOM (4 bytes) and any other overheads such as crypto headers or data required to flush the interleaver.

```
function modem_query_waveform()
      return curr_wf
      end
```

This function should return the current waveform or the empty table if there isn't one set yet.

```
function modem_transmit(data) ... end
```

Start a transmission of the given data, asynchronously. The `data` argument is the data of the entire transmission as a Lua string. As a result of this call, the entire transmission process should be scheduled, right through to dropping the carrier and unkeying the radio. The data is passed with a single call to avoid having to do flow control over this API, and to make flow control with the modem simpler (since the amount of data is already known).

The state, as updated with `modem_state()`, should raise `MODEM_STATE_TX_ACTIVE` when the modem reports that the transmission has started and then drop it and raise `MODEM_STATE_TX_READY` when the modem reports that it has finished.

```
require 'export_modem'
```

This statement should appear after the definitions above to export them and make them available to be called from outside this Lua VM.

## B.3 Modem to Server API

The `require 'modem'` statement imports the following functions specific to modem drivers, plus all those made available by `require 'std'` (see later sections):

```
modem_state(state)
```

Report the modem state back to the 5066 server. The `state` value is the sum of the `MODEM_STATE_*` values that apply at this moment. The driver should monitor the modem's status and report when reception and transmission starts and stops. The **hftool** can be used to check the timings of these changes.

**Table B.1. Modem state values: sum values that apply**

| | |
|---|---|
| `MODEM_STATE_OFFLINE` | Connection to modem down |
| `MODEM_STATE_ONLINE` | Connection to modem up |
| `MODEM_STATE_TERMINATED` | Modem driver terminated. (Set automatically on close or fatal error.) |
| `MODEM_STATE_TX_READY` | Ready to transmit |
| `MODEM_STATE_TX_ACTIVE` | Transmission in progress |
| `MODEM_STATE_RX_READY` | Ready to receive. Note that after switching to `RX_READY`, the modem may still be decoding the received signals and data may continue to be returned for some seconds afterwards. |
| `MODEM_STATE_RX_ACTIVE` | Reception in progress |
| `MODEM_STATE_CONFIG_READY` | Modem configuration is ready |
| `MODEM_STATE_CONFIGURING` | Modem is being configured |

```
modem_pdu(data)
```

Pass through data received from modem (as a Lua string).

```
modem_quality(table)
```

Pass through quality information about the currently active RX signal. Members understood within the table: `snr`: measured SNR in dB, `ber`: -log10 estimated BER, `bps` and `ilv`: BPS and interleaver of waveform currently being received. The modem driver should try and report whatever quality information is available from the modem at regular intervals whilst there is an incoming signal, to provide information to the rate-selection algorithm. If a piece of data is not available from the modem, it should be omitted from the table. The driver should poll the modem every few seconds for updated information.

```
modem_option_get(key)
```

Get the value associated with the given key from the modem config. `nil` is returned if it is not found. The commonly-used keys are as follows, although arbitrary keys and values may be put in the XML modem configuration (for Icon-5066) or command-line (for **hftool**) and received here:

**Table B.2. Keys for config options**

| | |
|---|---|
| `host` | Hostname (e.g. as provided to **-h** option in **hftool**) |
| `port` | Port number (e.g. as provided to **-p** option in **hftool**) |
| `chansim` | Present if this modem/device should set itself up as a channel simulator (assuming the modem/device supports that). The value is the preset name to use to configure the channel simulator. |

```
modem_conn_open(name, host, port, read_fn, state_fn)
```

Create a TCP connection to the given host and port, asynchronously. `name` is the name to associate with the connection. If there is already a connection with that name, no new connection is made and `false` is returned. Otherwise a connection is started and `true` is returned. `read_fn` is a Lua function that will be called to handle incoming data on this connection; it is called as `read_fn(data)`. `state_fn` is a Lua function that will be called to handle state changes on the connection; it is called as `state_fn(state)`, where state can be decomposed using the `is_tcp_*()` functions (see below)

```
modem_conn_close(name)
```

Close the connection with the given name, if it exists, and return `true`. Returns `false` if the named connection is not found.

```
modem_conn_send(name, data)
```

Send data on the named connection (asynchronously) and return `true`. Return `false` if the named connection was not found.

```
modem_conn_state(name)
```

Query the state of the connection, or return `0` if the named connection is not found. The returned state should be decomposed using the `is_tcp_*()` functions (see below).

```
is_tcp_connecting(state)
      is_tcp_running(state)
      is_tcp_stopped(state)
      is_tcp_destroy(state)
      is_tcp_processing(state)
```

The above functions check the TCP connection state returned by `modem_conn_state(name)` or passed to the `state_fn` callback function of `modem_conn_open()`. They each return a boolean value.

```
open_connection(name, host, port, read_fn, state_fn)
```

Open a connection (with `modem_conn_open()`, using the same arguments) and yield waiting for the connection-attempt to succeed or fail before returning. It uses a timer to timeout the connection attempt after 5 seconds if it takes too long. Returns: `true` on success, and `(false, msg)` on failure. This only works when called from a coroutine (since it runs asynchronously).

## B.4 Timer and utility functions

The following utility functions are made available in any Lua VM that does `require 'std'`:

```
timer_now()
```

Return the time in seconds since an arbitrary fixed time in the past to the best accuracy the platform supports.

```
timer_add(name, timeout, timeout_fn)
```

Set up a timer with the given name. If there is already a timer with the given name then the old timer will be cancelled first. The duration `timeout` is specified in seconds. When the timer completes, `timeout_fn` is called as `timeout_fn()`. The timer duration is handled to millisecond precision, but the callback may be delayed according to the processing load within the 5066 server.

```
timer_del(name)
```

Delete the timer with the given name.

```
warn(msg)
```

Write a warning message to the Isode logs. This can be helpful for debugging drivers.

```
hexdump(msg, data)
```

Write a warning message and a hexdump of the given data to the Isode logs. This can be helpful for debugging drivers.

```
crc_ccitt(data)
```

Calculate CCITT CRC of the data

```
crc(init, width, poly, reflect_input,
      reflect_crc, xor_crc, data)
```

Calculate CRC of the data (parameterised). See: http://www.ross.net/crc/download/crc_v3.txt for details of parameters, and http://reveng.sourceforge.net/crc-catalogue/all.htm for a catalogue of parameters for various CRCs.

```
pack(format, ...)
```

Pack the `...` arguments using the given format (documented below). The result is a binary string. If the format is exhausted and there are still more arguments then another format string is read from the arguments and processed. This allows the format string to be broken up into several parts, each with its own arguments following.

```
a,b,c,d,... = unpack(data, format, ...)
```

Unpack `data` using the given format. If there are `...` args, then these are taken as additional format strings. The format string(s) may fail to match the data, in which case the processing will stop at that point and a shortened list of read values will be returned, causing remaining values to be filled with nils. To be sure that the end of the data was reached, the `T` format character may be used (returning a boolean).

The format-string for packing and unpacking binary data may contain the following elements:

**Table B.3. Pack/unpack format-string elements**

| | |
|---|---|
| `' '` | Spaces are ignored |
| `[#]` | Repeat count, prefix to some other item |
| `##` | Byte as 2 hex digits, inserted on pack, matched on unpack |
| `s#` | Signed #-bit big-endian integer (8,16,24,32) to pack/unpack |
| `sr#` | Signed #-bit little-endian integer (8,16,24,32) to pack/unpack |
| `u#` | Unsigned #-bit big-endian integer (8,16,24,32) to pack/unpack |
| `ur#` | Unsigned #-bit little-endian integer (8,16,24,32) to pack/unpack |
| `x` | Ignored byte (for unpack only) |
| `T` | End-test (for unpack only): gives true if at end of data, false if not |
| (*bit-spec*) | Bit-packing. Contained spec must make up a whole number of bytes, which are read/written first-to-last, from MSB of first byte, to LSB of last byte. |

**Table B.4. Bit-packing format specification, inside parentheses**

| | |
|---|---|
| `' '` | Spaces are ignored |
| `0` | single bit with value 0, inserted on pack, matched on unpack |
| `1` | single bit with value 1, inserted on pack, matched on unpack |
| `x` | single bit ignored (for unpack only) |
| `s#` | #-bit signed value |
| `u#` | #-bit unsigned value |

## B.5 Coroutine support functions

Using coroutines simplifies coding of a sequence of actions which rely on callbacks to progress. A coroutine is *detached* from the main thread of execution, and then its execution may be paused and resumed as it progresses. The following functions are made available by `require 'std'`:

```
detach(func, ...)
```

Execute the given function with the given args as a coroutine. It will run up to the first `coroutine.yield()` before the `detach()` function returns. The rule is that whenever `coroutine.yield()` is called within this coroutine, the yielding code must have setup something somewhere to call `resume()` some time later, or else the coroutine will stop at that point and probably be GC'd due to there being no active references. Note that it is fine to detach one coroutine from another one. A reference to the created coroutine is returned.

```
corout()
```

Return a reference to the currently running coroutine, or cause a fatal error if this is the main thread.

```
resume(co, ...)
```

Resume the given coroutine with the given arguments. This is the same as `coroutine.resume()` in Lua except that errors are passed through.

```
sleep(dur)
```

Sleep for the given duration (in seconds, to ms accuracy) then resume the current coroutine. This only works from a coroutine.

## B.6 Waveform support functions

The `waveform` package contains waveform tables used to get lists of valid combinations of BPS, interleaver and bandwidth, and functions that can make timing calculations.

The STANAG 4539 and WBHF (MIL-STD-188-110C annex D) waveforms have parameters which may be overridden by setting global variables before the `waveform` package is pulled in with the `require` statement:

**Table B.5. Global variable parameters for waveform package**

| | |
|---|---|
| `WBHF_M_PARAM` | For WBHF: number of repeats of superframe, from 1 to 32. Time used is 240ms for each one, unless there is just one, in which case it is 133ms. |

| | |
|---|---|
| WBHF_N_PARAM | For WBHF: Length of TLC section, from 0 to 255. Time used is 13.33ms for each one. |
| S4539_PREAMBLE | For 4539: Length of TLC section, from 0 to 7. Time used is 77ms for each one. |

```
waveform.time_for_bytes(wf, len)
```

Return how long in seconds it will take to transmit the given number of bytes for the given waveform settings.

```
waveform.bytes_for_time(wf, dur)
```

Return how many bytes can be transmitted in the given duration in seconds for the given waveform settings.

```
waveform.list_waveforms(spec1, spec2)
```

List all the possible combinations of waveform settings that match the given match tables `spec1` and `spec2`. If either of the specification tables is missing it defaults to blank (which matches all). Each of the possible components of the `spec` table (wf, bps, ilv, bw) may be a precise value, or be a comma-separated list of values or ranges, or may be blank or missing (resulting in all possible values). A range is for matching numeric values, and takes the form of "*`digits-digits`*" or "*`digits`*:*`digits`*". The return value is a table containing a list of all valid waveform setting tables that match the specs.

```
waveform.filter(list, spec)
```

Return the given list of waveforms filtered by the given specification (as for `waveform.list_waveforms()`), if non-nil.

```
waveform.wbhf_index_for_bps(wf)
```

Return the BPS index in the WBHF tables (0-13) corresponding to the waveform specification passed. This is useful if the modem uses the same encoding as the WBHF standard.

## B.7 In-process loopback testing

There is support for creating a driver which simulates two or more modems within the Icon-5066 server via in-process broadcasts. This is used as the basis of the `loop` driver. It could potentially be used to simulate other aspects of idealised modem behaviour for testing without an external modem.

```
modem_broadcast_init()
```

Register this modem driver instance to receive broadcasts from other modem drivers in the same server process.

```
modem_broadcast(waveform, duration, data)
```

Broadcast a transmission to all other listening modem drivers. `waveform` should be a valid waveform-specification table. The other two parameters' intended purpose is indicated by their names, but how precisely they are interpreted is determinated by the specific implementation of the modem drivers.

```
modem_recv_broadcast(waveform, duration, data)
```

This global function must be implemented in the driver. Soon after `modem_broadcast(...)` is called in one driver, this function is called in all the registered drivers, excluding the one that made the broadcast, with the same arguments that were passed to `modem_broadcast()` above.