Bluetooth Architectural Overview and Configuration Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Monday, May 5, 2014

Table of Contents

About This Guide

The *Bluetooth Architectural Overview and Configuration Guide* describes the Bluetooth components supplied with the QNX CAR platform. This guide is intended for application developers who will be using Bluetooth technology in their in-car systems.

The following table may help you find information quickly:

To find out about:	Go to:
Our level of Bluetooth support	*Bluetooth Architecture* (p. 9)
Key components and how they interact	*Bluetooth Architecture* (p. 9)
Configuring the services in the Bluetooth startup sequence	*Starting Bluetooth on the Target* (p. 11)
Pairing and authentication	*Device Management* (p. 19)
Using the supported Bluetooth profiles (HFP, MAP, PBAP, SPP, and A2DP/AVRCP)	*Bluetooth Profiles* (p. 25)
Databases related to Bluetooth (core, phonebook, and messages)	*Bluetooth Databases* (p. 35)

Related documentation

The following references also contain relevant information on using Bluetooth with the QNX CAR platform:

- *PPS Objects Reference*—includes descriptions of several Bluetooth-related PPS objects used for issuing commands and for publishing status information
- WebWorks JavaScript Extensions (CAR 2.0—Deprecated) in the *HTML5 and JavaScript Framework*—includes descriptions of Bluetooth JavaScript extensions (`qnx.bluetooth`, `qnx.bluetooth.pbap`, and `qnx.bluetooth.spp`)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl –Alt –Delete**
Keyboard input	`Username`
Keyboard keys	`Enter`
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View** .

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Bluetooth Architecture

The QNX CAR Platform uses the Cybercom blueGO application software framework, which is a wrapper around Sybase iAnywhere's Bluetooth protocol stack and profiles.

Overview

The QNX CAR platform provides PPS objects for issuing Bluetooth commands and for interacting with profiles. QDB databases store messages and phonebook information read from a connected paired device. Both the QDB and PPS services interact with the `io-bluetooth` service, which talks to the Bluetooth hardware through the `devc-seromap_hci` driver.

For this release, we have tested Bluetooth on the Texas Instruments J5 ECO EVM811x EVM and OMAP5432 EVM boards. This is not to say that Bluetooth won't work on other hardware, but you'll need a BTS file created specifically to support your hardware's Bluetooth chip.

The following diagram shows the interaction between the main Bluetooth components:

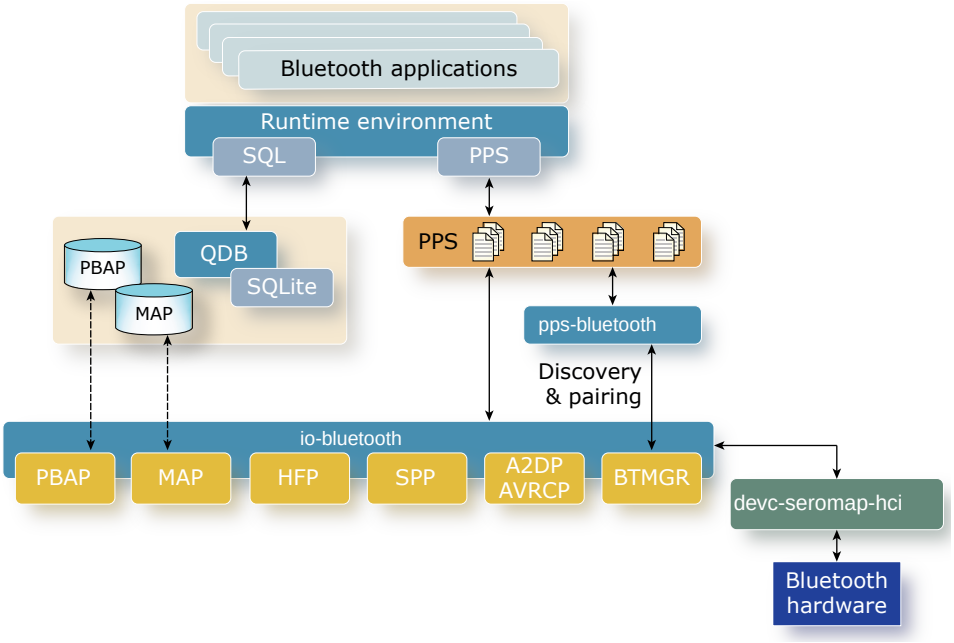

Figure 1: The Bluetooth architecture for the QNX CAR platform

Resource manager (`io-bluetooth`)

The front-end resource manager for Bluetooth is `io-bluetooth`, which offers a POSIX-compliant API and provides low-level access to the Bluetooth radio chip (typically

via a serial interface such as UART, USB, or I2S). The `io-bluetooth` manager also supports the Bluetooth profiles and provides profile concurrency and control.

PPS interface (`pps-bluetooth`)

The PPS abstraction layer for Bluetooth provides a filesystem-based interface for activities such as pairing devices, managing profiles, getting status updates, and so on. You'll find the following Bluetooth-related PPS objects on your system:

Core Bluetooth objects

- `/pps/services/bluetooth/control`
- `/pps/services/bluetooth/services`
- `/pps/services/bluetooth/settings`
- `/pps/services/bluetooth/status`
- `/pps/services/bluetooth/remote_devices/<mac_addr>`
- `/pps/services/bluetooth/paired_devices/<mac_addr>`

HFP-related objects

- `/pps/services/bluetooth/handsfree/control`
- `/pps/services/bluetooth/handsfree/status`

MAP-related objects

- `/pps/services/bluetooth/messages/control`
- `/pps/services/bluetooth/messages/notification`
- `/pps/services/bluetooth/messages/status`

PBAP-related objects

- `/pps/services/bluetooth/phonebook/control`
- `/pps/services/bluetooth/phonebook/status`

For details on using these PPS objects, see the *PPS Objects Reference*.

Serial driver (`devc-seromap_hci`)

The serial driver interfaces with the Bluetooth radio chip on the hardware. For details on starting this driver and other key processes, see "Starting Bluetooth on the Target" in this guide.

Chapter 2
Starting Bluetooth on the Target

The Bluetooth services on the QNX CAR platform depend on several system services that must be started in a certain sequence as outlined below.

Starting the required services

The following commands assume a Texas Instruments OMAP5432 EVM board.

1. Make sure the following general system services are running. Note that Bluetooth as well as many applications throughout the system rely on these services:
   - `pps`
   - `qdb`
   - `io-acoustic` (for HFP)
   - `mm-player` (for AVRCP)

   For more information on starting these services and their dependencies, see the `/etc/slm-config-all.xml` file on your target.

   

   To find out if a particular process is already running on your target, you can use the `pidin` utility (which displays information for all process IDs) and pipe the output through `grep`, specifying the process you're interested in. For example, to see if `io-acoustic` is running, use this command:

   ```
   # pidin | grep io-acoustic
   ```

2. Make sure the `devc-seromap_hci` driver (for HCI shared transport) is running. Here's the command line:

   ```
   devc-seromap_hci -E -f -a -g 0x4805b000,142 -c48000000/16
   0x48066000^2,137
   ```

3. Start the `io-bluetooth` service. Note that this command specifies the BTS file for the OMAP5432 board:

   ```
   io-bluetooth -vvvvv -s /etc/system/config/blue
   tooth/WL18xx_2.x_SP2.8.bts
   ```

4. Start the `pps-bluetooth` service:

   ```
   pps-bluetooth –vvvv
   ```

5. Start the following services, *if you need them*:

`bluetooth-map-initiator -vv` (for syncing messages)

`bluetooth-pbap-initiator -vv` (for syncing the phonebook)

`ifwatchd -A /scripts/ifarrv.sh -D /scripts/ifdepart.sh pan0` (PAN scripts)

If the `io-bluetooth` service isn't starting up, check the system log for multiple instances of `HCC_RESET failure`. If you find such errors, you'll need to slay and then restart the `devc-seromap_hci` driver. If you don't see those HCC-related errors, then make sure that the `pps-bluetooth` service is running, since it's the service that initializes `io-bluetooth`.

Command-line options for `devc-seromap_hci`

Although many general options are available for the `devc-seromap_hci` driver (because it's one of several drivers that rely on the `io-char` library), not all options make sense in the context of HCI. You *must* use the options as shown in **Step 2** above to enable these specific features, but *not* their counterparts:

- *raw* input mode (`-E`)
- hardware flow control (`-f`)
- auto-RTS (`-a`)

For instance, don't use *edited* mode—this would break HCI packet handling!

Here are the command-line options you can use when starting the `devc-seromap_hci` driver:

`devc-seromap_hci [`*`options`*`] [`*`port`*`[^`*`shift`*`][,`*`irq`*`][,k]] &`

`-a`

Use auto-RTS when hardware flow control is enabled.

`-b` ***number***

Define initial baud rate (default is 115200).

Always start the driver with the default baud rate (115200). The BT chip will default to this rate when it comes out of reset, so the UART must be configured for this rate to issue the initial commands. If you want to increase the baud rate, the commands should be sent by the BT stack *after* the driver has reported CARRIER. The sequence is as follows:

1. The BT stack writes the HCI commands for a change in baud rate to the BT chip via the HCI driver.
2. The stack waits for the acknowledgment.
3. The stack then changes the local HCI driver baud rate via the *tcsetattr()* call.

This sequence can be included in the `*.bts` file so that the value of the new baud rate won't need to be hard-coded into the BT stack.

`-c` *clk*[*/div*]

Set input clock rate (in hertz) and divisor.

`-C` *number*

Set size of canonical input buffer (default is 256 bytes).

`-e`

Set options to *edited* mode.

`-E`

Set options to *raw* mode (default).

`-f`

Enable hardware flow control (default).

`-F`

Disable hardware flow control.

`-g` *gpio_base,gpio_pin*

GPIO base and GPIO pin used for Bluetooth Enable. On driver initialization or UART transmit/receive error, the driver will toggle the GPIO connected to the chip's BT_EN or BT_RST pin and then issue the HCC_RESET command.

`-I` *number*

Set size of raw input buffer (default is 2048 bytes).

`-l (0|1)`

("el") Enable *loopback* mode (`1`=on, `0`=off).

`-n`

Name of Bluetooth Script. If you use the `-n` option, the driver will follow up the HCC_RESET command by uploading the provided file (`*.bts`). Note

that the `-n` is optional since the Bluetooth stack sitting above the HCI UART driver can open the file directly and write the encoded HCI commands down to the driver via regular *write()* calls.

The driver will use the CD (Carrier Detect) line status to communicate when the interfaces are "ready" to be used, i.e., the driver will report CARRIER to the client, then the HCC_RESET and script load will be successfully completed. If the `-n` option isn't used to enable script load, then CARRIER is reported *after* the HCC_RESET. Note that the carrier signal is reported as dropped/lost when we reset the chip by toggling the GPIO pin.

`-O` *number*

Set size of output buffer (default is 2048 bytes).

`-s`

Enable software flow control.

`-S`

Disable software flow control (default).

`-t` *number*

Set receive FIFO trigger level (default is 16).

`-T` *number*

Set transmit FIFO trigger level (default is 8).

`-u` *unit*

Set serial unit number (default is 1).

`-U` *uid:gid*

Set the user ID and group ID.

`-v`

Be verbose. Use multiple `v`'s to increase verbosity (see `sloginfo` data).

port

Hex I/O address of serial port.

shift

The spacing of the device registers as a power of 2 (e.g., `0` means registers are 1 byte apart, `1` means registers are 2 bytes apart, etc.). The default *shift* is `0`.

irq

Interrupt used by the port. To specify in hex, prefix with `0x`.

k

Place this after the *irq* value to indicate that a Maxim RS-232 transceiver is used on this port, which requires sending it a null character to wake it up after going into *Autoshutdown Plus* mode.

By default, the CREAD terminal flag will be disabled for all HCI interfaces (`serbtX`, `sergpsX`, and `serfmX`). The Bluetooth stack (or whatever app wants to use these interfaces) must set the CREAD flag to enable receive functionality (on a per-interface basis). For proper functionality, the client app must clear the flag when finished with the device/interface. This is because all interfaces share the same UART hardware—we don't want to get stuck in a flow-controlled state because there's no client reading from one of the interfaces. Flow control will be asserted when any of the device buffers reaches the defined high-water mark and won't be cleared until there's room in *all* interface buffers to receive more data. In other words, if the system doesn't care about GPS and there's no client reading from the `sergpsX` interface, we don't want the driver to buffer GPS data, which would eventually fill the buffer and assert the flow-controlled state.

Command-line options for `io-bluetooth`

Here are the command-line options you can use when starting the `io-bluetooth` service:

`io-bluetooth [-d` *`path`*`][-f` *`FD`*`][-r` *`file`*`][-s` *`path`*`][-v]`

`-d` ***path***

Set the path to the serial driver (default is `/dev/serbt1`).

`-f` ***FD***

Specify the file descriptor for the log destination (e.g., `2` for *stderr*).

`-r` ***file***

Dump raw data to the specified text file. The relevant profile will be added to your filename after an underscore (i.e., *`file_profile`*`.txt`). For example, if you use `-r /mydata`, the result will create `/mydata_map.txt` and `/mydata_pbap.txt`.

`-s` ***path***

Set the path to the BTS file (default is `/etc/system/config/bluetooth/WL127x_2.0_SP1.bts`). This file informs the hardware of the low-level parameters of the Bluetooth exchange, including the line numbers to use and the baud rate.

`-v`

Be verbose. Use multiple `v`'s to increase verbosity (see `sloginfo` data).

Command-line options for `pps-bluetooth`

Here are the command-line options you can use when starting the `pps-bluetooth` service:

```
pps-bluetooth [-f FD][-n name] [-U euid:egid][-v]
```

`-f` ***`FD`***

Specify the file descriptor for the log destination (e.g., `2` for *stderr*).

`-n` ***`name`***

Set the local name of the Bluetooth device.

`-U` ***`euid`*****`:`** ***`egid`***

Specify the effective user ID and group ID of the `pps-bluetooth` process.

`-v`

Be verbose. Use multiple `v`'s to increase verbosity (see `sloginfo` data).

Command line for `bluetooth-map-initiator`

The `bluetooth-map-initiator` executable (for syncing messages) takes only one option:

```
bluetooth-map-initiator [-v]
```

`-v`

Be verbose. Use multiple `v`'s to increase verbosity (see `sloginfo` data).

Command line for `bluetooth-pbap-initiator`

The `bluetooth-pbap-initiator` executable (for syncing the phonebook) takes only one option:

```
bluetooth-pbap-initiator [-v]
```

`-v`

Be verbose. Use multiple `v`'s to increase verbosity (see `sloginfo` data).

Automated startup via SLM

The System Launch and Monitor (SLM) is a utility used for automating the startup sequence of processes and any interprocess dependencies. SLM itself is started early in the boot sequence (from `startup.sh`) to launch complex applications consisting of many processes that must be started in a specific order. For more information, see the entry for `slm` in the OS *Utilities Reference*.

Here are the relevant sections from the `/etc/slm-config-platform.xml` configuration file for starting variant-specific processes (in this case for an OMAP5432 board):

HCI shared transport serial driver:

```
<SLM:component name="hci">
    <SLM:command>devc-seromap_hci</SLM:command>
 <SLM:args>-E -f -a -g 0x4805b000,142 -c48000000/16 0x48066000^2,137</SLM:args>
 <SLM:waitfor wait="pathname">/dev/serbt1</SLM:waitfor>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
</SLM:component>
```

The `io-bluetooth` service:

```
<SLM:component name="bluetooth">
    <SLM:command>io-bluetooth</SLM:command>
 <SLM:args>-vvvvv -s /etc/system/config/bluetooth/WL18xx_2.x_SP2.8.bts</SLM:args>
 <SLM:waitfor wait="pathname">/dev/io-bluetooth/btmgr</SLM:waitfor>
 <SLM:stdout>/var/log/io-bluetooth/stdout</SLM:stdout>
 <SLM:stderr>/var/log/io-bluetooth/stderr</SLM:stderr>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
 <SLM:depend>hci</SLM:depend>
 <SLM:depend>ioacoustic</SLM:depend>
 <SLM:depend>qdb</SLM:depend>
</SLM:component>
```

The `pps-bluetooth` service:

```
<SLM:component name="pps-bluetooth">
    <SLM:command>pps-bluetooth</SLM:command>
 <SLM:args>-vvvvv</SLM:args>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
 <SLM:depend>bluetooth</SLM:depend>
 <SLM:depend>pps</SLM:depend>
</SLM:component>
```

Phonebook sync automator:

```
<SLM:component name="pps-pbap-initiator">
    <SLM:command>bluetooth-pbap-initiator</SLM:command>
 <SLM:args>-vv</SLM:args>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
 <SLM:depend>pps</SLM:depend>
</SLM:component>
```

Messages sync automator:

```
<SLM:component name="pps-map-initiator">
    <SLM:command>bluetooth-map-initiator</SLM:command>
 <SLM:args>-vv</SLM:args>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
 <SLM:depend>pps</SLM:depend>
 <SLM:depend>qdb</SLM:depend>
</SLM:component>
```

PAN scripts to start and stop `dhcp.client` to get an IP from the phone:

```
<SLM:component name="pan-if-monitor">
    <SLM:command>ifwatchd</SLM:command>
 <SLM:args>-A /scripts/ifarrv.sh -D /scripts/ifdepart.sh pan0</SLM:args>
 <SLM:stop stop="signal">SIGTERM</SLM:stop>
</SLM:component>
```

Chapter 3
Device Management

The QNX CAR platform supports three Bluetooth device operations: pairing a device (e.g., a car's head unit or a smartphone), removing a paired device from the system, and obtaining status information for devices.

Pairing a device

To be able to connect and transfer data between two devices (such as the car's head unit and a smartphone), you have to go through the *pairing* process. Pairing necessarily involves authentication so as to ensure security when connecting.

Pairing interactions

The following diagram shows the interactions involving the HMI, the PPS interface (`pps-bluetooth`), and the Bluetooth Manager (`io-bluetooth`):

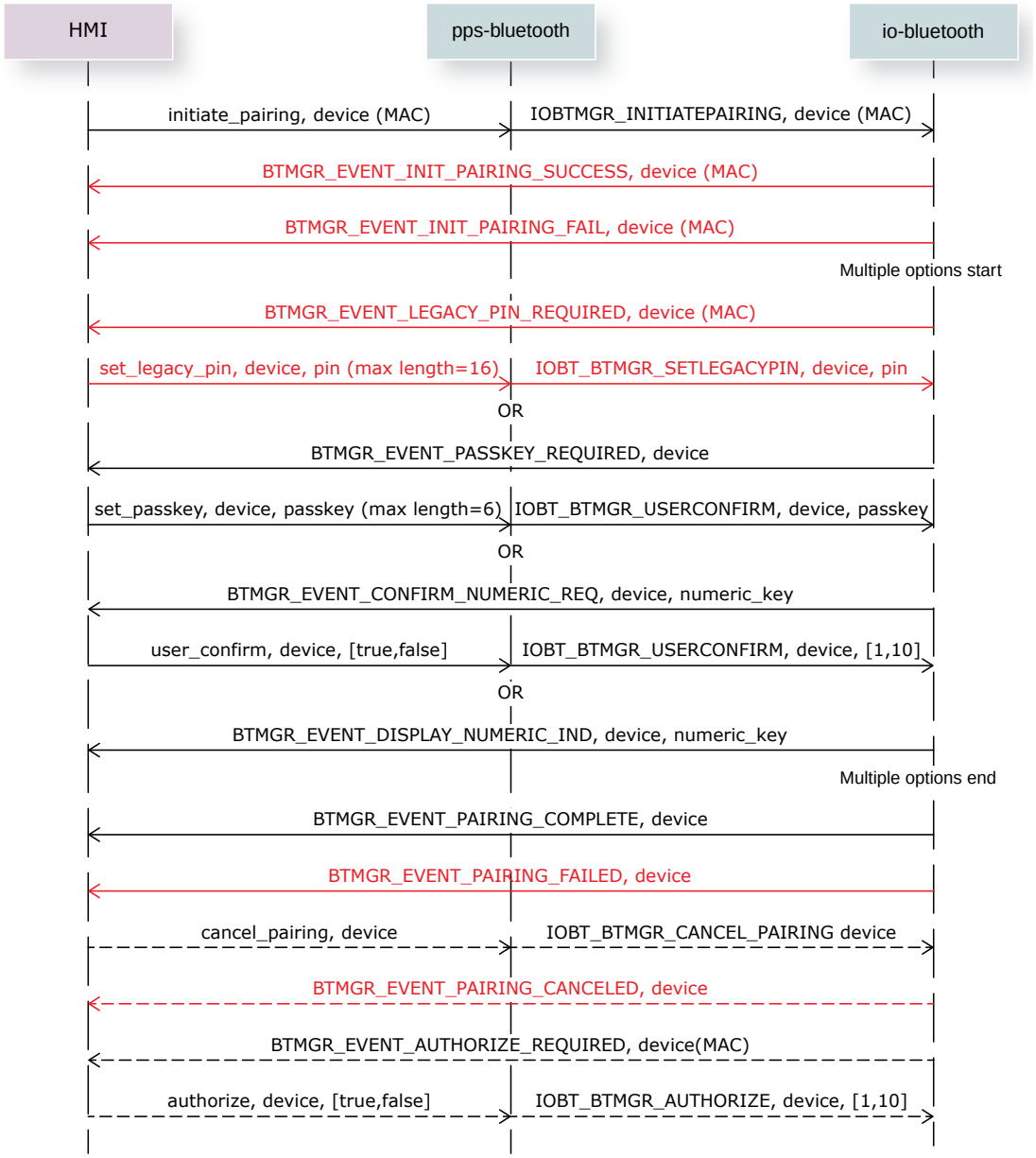

Figure 2: Typical message exchange between the HMI, `pps-bluetooth`, and `io-bluetooth`

The Bluetooth Manager listens for commands such as `initiate_pairing` on the `/pps/services/bluetooth/control` object and then publishes `BTMGR_EVENT_*` events to the `/pps/services/bluetooth/status` object.

Adding a device

When the HMI receives a request to add a device (e.g., the user taps **ADD NEW DEVICE** in the Bluetooth Connectivity screen under **Settings** in the HMI), the Bluetooth Manager will issue a `BTMGR_EVENT_DEVICE_ADDED` event to the `pps-bluetooth` service, which will publish the appropriate status information to these PPS objects:

- `/pps/services/bluetooth/remote_devices/<mac_addr>`
- `/pps/services/bluetooth/status`

Making devices discoverable

The `set_access` Bluetooth command lets you set the accessibility level of the Bluetooth system as follows:

Number value in *data* parameter:	Meaning:
`0`	Devices won't be discovered or connected (`IOBT_NOT_ACCESSIBLE`)
`1`	Devices may be discovered and connected (`IOBT_GENERAL_ACCESSIBLE`)
`2`	Devices will have limited discoverability and connectability (`IOBT_LIMITED_ACCESSIBLE`)
`3`	Devices may be connected, but not discovered (`IOBT_CONNECTABLE_ONLY`)
`4`	Devices may be discovered, but not connected (`IOBT_DISCOVERABLE_ONLY`)

For example, the following `set_access` command will set the accessibility level to `1` (so devices may be discovered and connected):

```
echo "command::set_access\n
      data:n:1" >>
      /pps/services/bluetooth/control
```

Removing a paired device

When a device is deleted (i.e., the user taps **DELETE** in the Bluetooth Connectivity control in the **Settings** screen), the HMI publishes a `remove_device` command containing the device's MAC address in the *data* parameter to the `/pps/services/bluetooth/control` object, which is read by `pps-bluetooth`. This service then issues an `IOBT_BTMGR_REMOVEDEVICE` event to the Bluetooth manager (`io-bluetooth`) and also publishes a status update to the `/pps/services/bluetooth/status` object.

The `/pps/services/bluetooth/paired_devices/` directory stores a PPS object for each successfully paired device. When a device is removed, its object (named after its MAC address) is deleted from this directory.

Getting device information

The `io-bluetooth` manager publishes status information for Bluetooth devices to the following PPS objects:

This PPS object:	Contains:
`/pps/services/bluetooth/messages/notification`	Status of messages per *account_id* from the MAP database.
`/pps/services/bluetooth/messages/status`	The results of commands sent to the `/pps/services/bluetooth/messages/control` object.
`/pps/services/bluetooth/paired_devices/<mac_addr>`	For each paired device, the profile services available, COD, etc.
`/pps/services/bluetooth/services`	The profiles used for a connected device.
`/pps/services/bluetooth/phonebook/status`	MAC address, state, and status info for devices connecting via PBAP.
`/pps/services/bluetooth/remote_devices/<mac_addr>`	For each discovered device, the profile services available, COD, etc.
`/pps/services/bluetooth/settings`	Stack info, such as active connections, MAC address of the local Bluetooth chip, etc.
`/pps/services/bluetooth/status`	Events in response to commands sent to the `/pps/services/bluetooth/control` object.
`/pps/services/bluetooth/handsfree/status`	The results of commands sent to the `/pps/services/bluetooth/handsfree/control` object.

Chapter 4
Bluetooth Profiles

A Bluetooth connection can be initiated either by the vehicle's head unit or by a mobile device. For each requested connection, you must select a Bluetooth profile based on which operations you want to perform.

After the connection has been established, the system creates PPS objects that `io-bluetooth` uses to manage the connection according to the permissions and other parameters in the selected profile.

The QNX CAR platform currently supports these profiles:

- HFP v1.4 (for making handsfree calls on connected devices)
- MAP v1.3.1 (for accessing email and text messages on devices)
- PBAP v1.4.1 (for accessing contact information on devices)
- SPP v1.1 (for exchanging raw data between the head unit and devices)
- A2DP/AVRCP v1.3 (for playing media)

Profile architecture

The following diagram shows the components involved with the operation of the HFP, PBAP, and MAP profiles. Each of these profiles has a control object to accept commands from HMI apps as well as a status object to report command results and the state of the Bluetooth service. The profiles run within `io-bluetooth`, which subscribes to the control objects and publishes to the status and notification objects. Only the MAP profile uses a notification object, which stores information on messages received.

The PBAP and MAP profiles modify information in their QDB databases, which the HMI can read. Here, the term *modify* refers to the SQL operations of `CREATE`, `INSERT`, `UPDATE`, and `DELETE`.

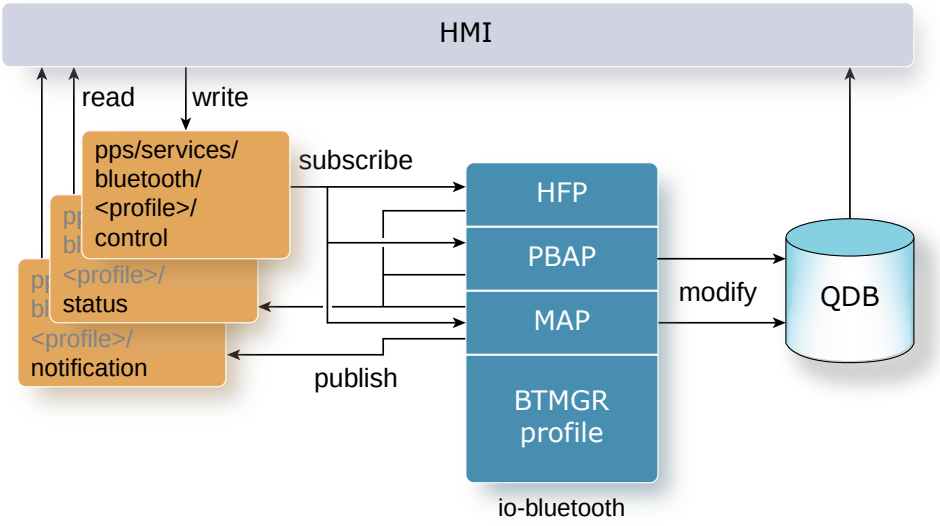

Figure 3: The components involved with the operation of the HFP, PBAP, and MAP profiles

The SPP and AVRCP profiles don't follow the same design of using separate PPS objects for accepting commands and for reporting their outcomes. Apps that need to stream data over SPP may choose to use PPS. For example, the `pps-spp` service, which supports HTML5 applications that need to access Bluetooth SPP data, uses the `/pps/services/bluetooth/spp/spp` object. AVRCP is controlled through a C API and doesn't directly use PPS objects. See the descriptions of these two profiles for information on how they interact with other components.

Hands-Free Profile (HFP)

HFP allows the car head unit to communicate with mobile phones in the car.

Connecting

To connect via HFP, simply send the **`connect_service`** command with the MAC address as the *data* parameter and the profile number (`0x111E`) as *data2* to the `/pps/services/bluetooth/control` object. For example:

```
echo "command::connect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x111E" >>
      /pps/services/bluetooth/control
```

In response to commands sent to the control object, the Bluetooth Manager publishes appropriate events (e.g., `BTMGR_EVENT_CONNECT_ALL_SUCCESS`) in the `/pps/services/bluetooth/status` object.

Using HFP

With the HFP profile, you can initiate a call, accept an incoming call, or terminate an active call.

To perform any of these actions, you must write the appropriate command (e.g., `HFP_CALL`) to the `/pps/services/bluetooth/handsfree/control` object.

Our HFP implementation supports only one call at a time.

Reading HFP status

You can read the `/pps/services/handsfree/status` object to learn the outcome of the last HFP command and the call state of the paired mobile device. This last field tells you if the device's phone line is idle or in use and if a call is on hold, being initialized, or already connected.

Disconnecting

To disconnect, simply send the **`disconnect_service`** command using the same parameters you used to connect. For example:

```
echo "command::disconnect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x111E" >>
      /pps/services/bluetooth/control
```

Message Access Profile (MAP)

MAP supports the exchange of messages between paired devices.

The MAP profile allows you to read the SMS, MMS, and email content on a connected mobile device from the head unit. For a mobile device (e.g., a smartphone), two accounts are typically available:

- a single aggregated SMS/MMS account tied to the device's phone number
- a corporate/personal email account

Each of these accounts will be entered into the *accounts* table of the messages database. Note that you'll need to reference the *account_id* when making any requests.

Connecting

To connect via MAP, simply send the **`connect_service`** command with the MAC address as the *data* parameter and the profile number (`0x1134`) as *data2* to the `/pps/services/bluetooth/control` object. For example:

```
echo "command::connect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1134" >>
      /pps/services/bluetooth/control
```

In response to commands sent to the control object, the Bluetooth Manager publishes appropriate events (e.g., `BTMGR_EVENT_CONNECT_ALL_SUCCESS`) in the `/pps/services/bluetooth/status` object.

Using MAP

The MAP profile allows you to browse an account's folder. You can view the messages listed and fetch the one you want.

Our MAP implementation currently has these limitations:

- You can't send messages
- You can sync only the default mail folders, not any nested folders
- You can sync only 100 messages per folder per account

You can also mark messages as *read* or *unread* or you can delete them.

To perform any of these actions, you must write the appropriate command to the `/pps/services/bluetooth/messages/control` object.

Monitoring messaging activity

Besides browsing an account's folder, you can read the `/pps/services/bluetooth/messages/notification` object to know when a new message is received, when a message is deleted, when a message is moved to a different folder, and other details.

Reading MAP status

You can read the `/pps/services/bluetooth/messages/status` object to learn the paired device's connection state and the profile's command-processing status (i.e., whether a command is currently being processed and the outcome of the last MAP command).

Automated initiator

An automated initiator program (`bluetooth-map-initiator`) will sync the MAP profile before its state will transition to `connected`. For each email account, the initiator will sync the first 100 messages from these folders:

- `inbox`
- `outbox`
- `deleted`
- `sent`

Disconnecting

To disconnect, simply send the **`disconnect_service`** command using the same parameters you used to connect. For example:

```
echo "command::disconnect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1134" >>
      /pps/services/bluetooth/control
```

Phone Book Access Profile (PBAP)

PBAP supports the exchange of Phone Book Objects between devices.

The PBAP profile allows you to sync the contact information on the remote device with the head unit's Bluetooth system. PBAP automatically downloads the call history (log of incoming calls, outgoing calls, and missed calls) when the mobile phone is first connected.

Connecting

To connect via PBAP, simply send the **`connect_service`** command with the MAC address as the *data* parameter and the profile number (`0x1130`) as *data2* to the `/pps/services/bluetooth/control` object. For example:

```
echo "command::connect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1130" >>
      /pps/services/bluetooth/control
```

In response to commands sent to the control object, the Bluetooth Manager publishes appropriate events (e.g., `BTMGR_EVENT_CONNECT_ALL_SUCCESS`) in the `/pps/services/bluetooth/status` object.

Using PBAP

The profile allows you to send it one command: **`SYNC_START`**. This command will erase the database and repopulate it with fresh data received from the device.

To perform this action, you must write the command to the `/pps/services/bluetooth/phonebook/control` object.

Reading PBAP status

You can read the `/pps/services/bluetooth/phonebook/status` object to know whether a particular device is connected, whether any error occurred during a connection attempt, and so on.

PBAP interactions

The following diagram shows the interactions involving the HMI, PPS, and PBAP:

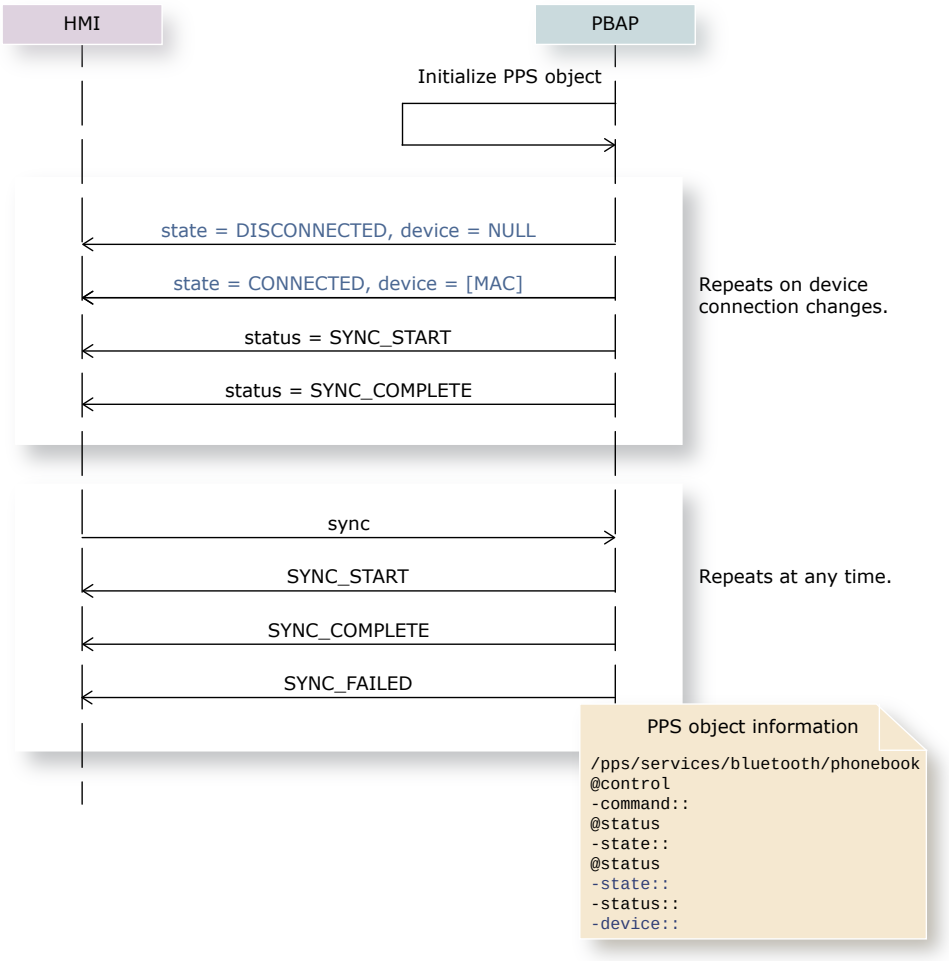

Figure 4: Typical PPS message exchange between the HMI and the PBAP profile

Automated initiator

An automated initiator program (`bluetooth-pbap-initiator`) will sync the PBAP profile before its state will transition to `connected`.

Disconnecting

To disconnect, simply send the **`disconnect_service`** command using the same parameters you used to connect. For example:

```
echo "command::disconnect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1130" >>
      /pps/services/bluetooth/control
```

Serial Port Profile (SPP)

SPP emulates an RS-232 serial connection, thereby supporting raw binary communication between two Bluetooth devices.

Connecting

To connect via SPP, simply send the **`connect_service`** command with the MAC address as the *data* parameter and the profile number (`0x1101`) along with the UUID of the SPP server you wish to connect to as *data2* to the `/pps/services/bluetooth/control` object. For example:

```
echo "command::connect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1101:5DF26DC6-8E42-8401-6D98-75C100B108B1" >>
      /pps/services/bluetooth/control
```

Using SPP

When reading from and writing to the remote device, see the following files for file descriptors:

`/dev/io-bluetooth/spp/UUID/stdin`

`/dev/io-bluetooth/spp/UUID/stdout`

You can use regular filesystem read/write facilities to read from or write to these mount paths. So if you're connected to SPP, you can use commands such as **`cat`** for reading or **`echo`** for writing to these paths as a way to get and send data from an SPP-connected phone.

Disconnecting

To disconnect, simply send the **`disconnect_service`** command using the same parameters you used to connect. For example:

```
echo "command::disconnect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x1101:5DF26DC6-8E42-8401-6D98-75C100B108B1" >>
      /pps/services/bluetooth/control
```

Advanced Audio Distribution Profile / Audio/Video Remote Control Profile (A2DP/AVRCP)

AVCRP allows the head unit to control media playback on connected devices.

The AVRCP profile is used with the Advanced Audio Distribution Profile (A2DP) profile, which supports streaming of high-quality audio, in mono and stereo, from a mobile device to the head unit.

Architecture

The following diagram shows the components involved with the operation of the AVRCP profile.

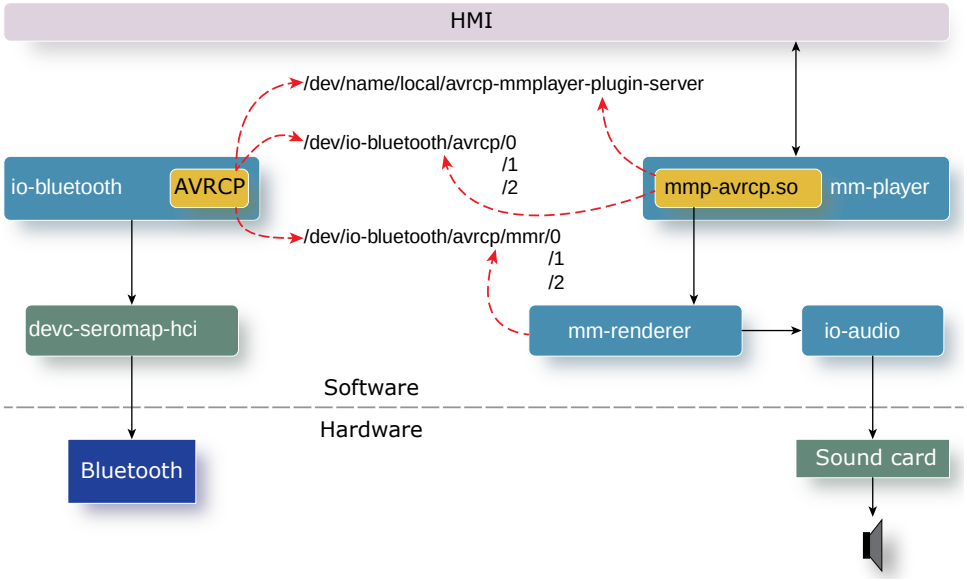

Figure 5: The components involved with the operation of the AVRCP profile

The AVRCP profile writes playback status updates to `/dev/name/local/avrcp-mmplayer-plugin-server`, creates the pathnames for the media player devices (`/dev/io-bluetooth/avrcp/#`, where # is an integer), and then monitors these entries for new playback commands. It also writes the audio data read from the Bluetooth hardware and `devc-seromap_hci` serial driver to the audio data device entries (`/dev/io-bluetooth/avrcp/mmr/#`, where # is an integer).

The `mmp-avrcp.so` plugin supports AVRCP. The `mm-player` service uses this plugin to forward playback commands, issued by the user in the HMI, to the appropriate media player device path and to `mm-renderer`. This last service reads media streams through the audio data device entries and then sends these streams to `io-audio`, which outputs the audio through hardware. The plugin also reads status information from the `avrcp-mmplayer-plugin-server` device entry and updates the HMI with this information as needed.

Connecting

To connect via A2DP/AVRCP, simply send the **`connect_service`** command with the MAC address as the *data* parameter and the profile number (`0x110B`) as *data2* to the `/pps/services/bluetooth/control` object. For example:

```
echo "command::connect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x110B" >>
      /pps/services/bluetooth/control
```

In response to commands sent to the control object, the Bluetooth Manager publishes appropriate events (e.g., `BTMGR_EVENT_CONNECT_ALL_SUCCESS`) in the `/pps/services/bluetooth/status` object.

A2DP/AVRCP works only with the `mm-player` media service; it doesn't work with the legacy `mm-control` service.

Disconnecting

To disconnect, simply send the **`disconnect_service`** command using the same parameters you used to connect. For example:

```
echo "command::disconnect_service\n
      data::BA:C3:32:AD:55:CC\n
      data2::0x110B" >>
      /pps/services/bluetooth/control
```

Chapter 5
Bluetooth Databases

The QNX CAR platform uses the following Bluetooth databases:

Storage file	Description
`bluetoothdb.db`	Core database—contains authentication data for connecting to devices.
`phonebook.db`	Phonebook database—contains PBAP-specific data.
`messages.db`	Messages database—contains MAP-specific data.

Each Bluetooth database has a raw SQLite storage file (`.db`) and a schema file (`.sql`) that defines the schema for creating the database. The phonebook and messages databases each have an additional `.sql` file that populates the database with initial data.

Database backups

The system keeps *two* backup copies of the core database (`bluetoothdb.db`) in these directories:

- `/var/db/backup/`
- `/var/db/backup2/`

A backup is made whenever a device is paired or connected. Each backup overwrites the older of the two existing backup files. This policy ensures that if the system crashes after many devices have been paired, none of these devices will have to be paired again when the system reboots because their authentication data can be recovered from the backups.

Core database

The core database contains all the authentication information needed to connect or reconnect to a device.

ER diagram

The following entity-relationship (ER) diagram shows the relationships among the tables in the core database:

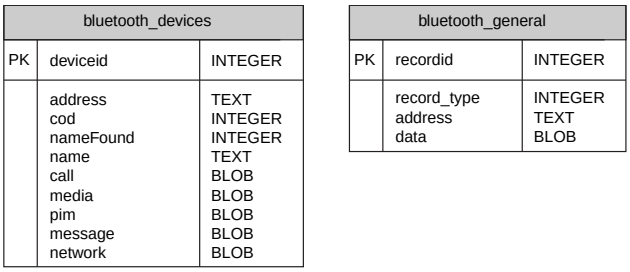

Figure 6: ER diagram for core database

Sample schema file

The database schema is specified in `/db/bluetoothdb.sql`, which looks like this:

```
CREATE TABLE bluetooth_general(
    recordid INTEGER PRIMARY KEY AUTOINCREMENT,
    record_type INTEGER,
    address TEXT,
    data BLOB
);

CREATE TABLE bluetooth_devices(
    deviceid INTEGER PRIMARY KEY AUTOINCREMENT,
    address TEXT,
    cod INTEGER,
    nameFound INTEGER,
    name TEXT,
    call BLOB,
    media BLOB,
    pim BLOB,
    message BLOB,
    network BLOB
);
```

Phonebook database

The phonebook database contains all the PBAP-specific data.

ER diagram

The following entity-relationship (ER) diagram shows the relationships among the tables in the phonebook database:

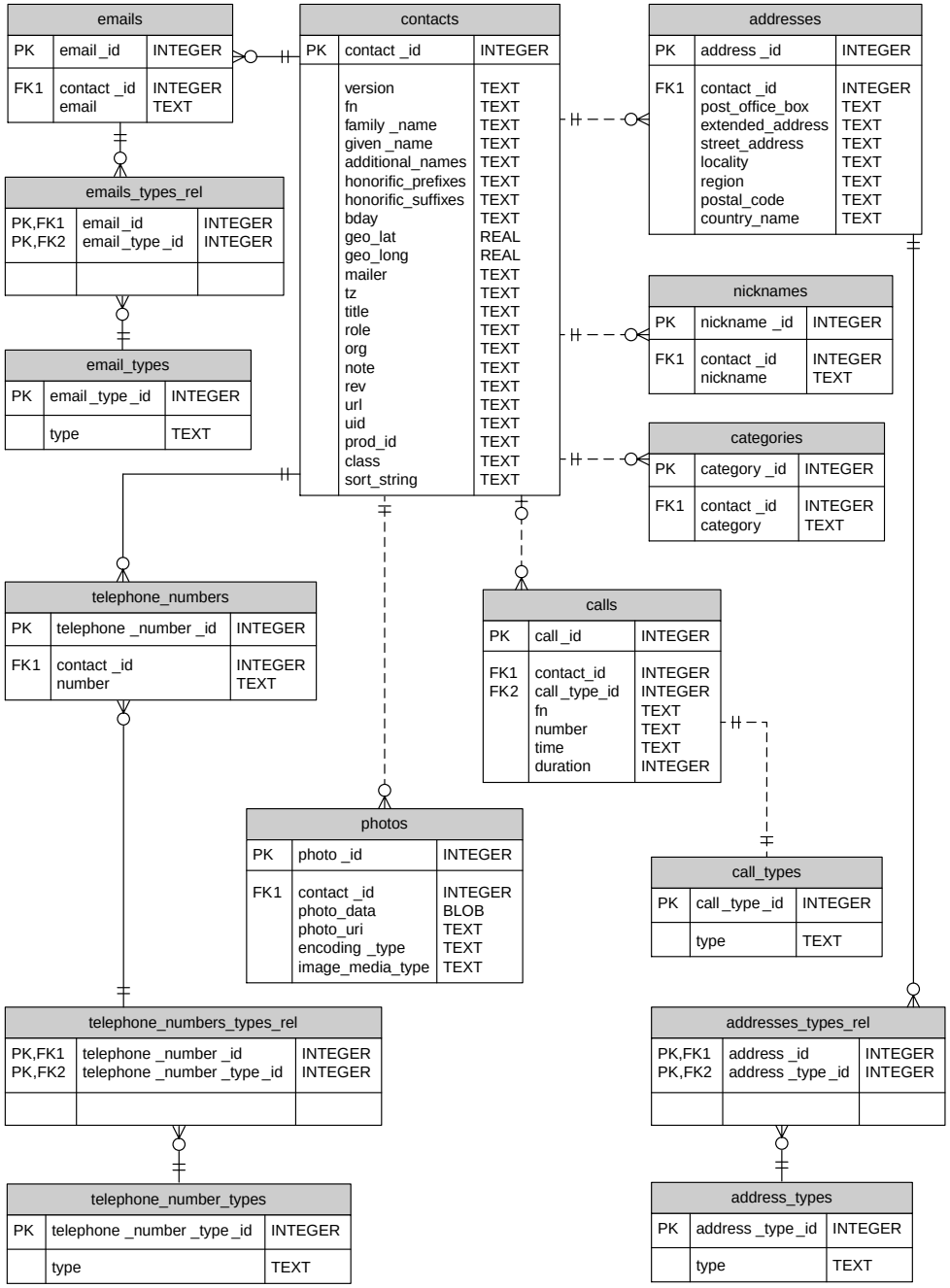

Figure 7: ER diagram for phonebook database

Sample schema file

The database schema is specified in `/db/phonebook.sql`, which looks like this:

```
/* Turn foreign key constraints on */
PRAGMA foreign_keys = ON;

/* Contacts */
CREATE TABLE contacts (
 contact_id   INTEGER PRIMARY KEY AUTOINCREMENT,
 version    TEXT NOT NULL,
 fn     TEXT NOT NULL,
 family_name   TEXT NOT NULL,
 given_name   TEXT NOT NULL,
 additional_names TEXT,
 honorific_prefixes TEXT,
 honorific_suffixes TEXT,
 sort_string   TEXT,
 bday    TEXT,
 geo_lat    REAL,
 geo_long   REAL,
 mailer    TEXT,
 tz     TEXT,
 title    TEXT,
 role    TEXT,
 org     TEXT,
 note    TEXT,
 rev     TEXT,
 url     TEXT,
 uid     TEXT,
 prod_id    TEXT,
 class    TEXT
);

/* Emails */
CREATE TABLE emails (
 email_id INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 email  TEXT NOT NULL
);

CREATE TABLE email_types (
 email_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
 type   TEXT NOT NULL UNIQUE
);

CREATE TABLE emails_types_rel (
 email_id  INTEGER NOT NULL REFERENCES emails ON DELETE CASCADE,
 email_type_id INTEGER NOT NULL REFERENCES email_types
);

/* Telephone numbers */
CREATE TABLE telephone_numbers (
 telephone_number_id INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id   INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 number    TEXT NOT NULL
);

CREATE TABLE telephone_number_types (
 telephone_number_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
 type      TEXT NOT NULL UNIQUE
);

CREATE TABLE telephone_numbers_types_rel (
 telephone_number_id   INTEGER NOT NULL REFERENCES telephone_numbers ON DELETE CASCADE,
 telephone_number_type_id INTEGER NOT NULL REFERENCES telephone_number_types
```

```
);

/* Addresses */
CREATE TABLE addresses (
 address_id   INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id   INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 post_office_box  TEXT,
 extended_address TEXT,
 street_address  TEXT,
 locality   TEXT,
 region    TEXT,
 postal_code   TEXT,
 country_name  TEXT
);

CREATE TABLE address_types (
 address_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
 type   TEXT NOT NULL UNIQUE
);

CREATE TABLE addresses_types_rel (
 address_id  INTEGER NOT NULL REFERENCES addresses ON DELETE CASCADE,
 address_type_id INTEGER NOT NULL REFERENCES address_types
);

/* Nicknames */
CREATE TABLE nicknames (
 nickname_id INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 nickname TEXT NOT NULL
);

/* Categories */
CREATE TABLE categories (
 category_id INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 category TEXT NOT NULL
);

/* Photos */
CREATE TABLE photos (
 photo_id   INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id   INTEGER NOT NULL REFERENCES contacts ON DELETE CASCADE,
 photo_data   BLOB,
 photo_uri   TEXT,
 encoding_type  TEXT,
 image_media_type TEXT
);

/* Call Log */
CREATE TABLE calls (
 call_id   INTEGER PRIMARY KEY AUTOINCREMENT,
 contact_id  INTEGER REFERENCES contacts ON DELETE SET NULL,
 call_type_id INTEGER NOT NULL REFERENCES call_types,
 fn    TEXT,
 number   TEXT,
 time   TEXT NOT NULL,
 duration  INTEGER
);

CREATE TABLE call_types (
 call_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
 type   TEXT
);
```

```
/********
  VIEWS
********/
CREATE VIEW emails_view AS
SELECT
 emails.email_id,
 emails.contact_id,
 emails.email,
 MAX(CASE WHEN email_types.type = 'PREF' THEN 1 ELSE 0 END) AS pref,
 MAX(CASE WHEN email_types.type = 'INTERNET' THEN 1 ELSE 0 END) AS internet
FROM emails
LEFT JOIN emails_types_rel ON emails.email_id = emails_types_rel.email_id
LEFT JOIN email_types ON emails_types_rel.email_type_id = email_types.email_type_id
GROUP BY emails.email_id
ORDER BY pref DESC, emails.email_id DESC;

CREATE VIEW telephone_numbers_view AS
SELECT
 telephone_numbers.telephone_number_id,
 telephone_numbers.contact_id,
 telephone_numbers.number,
 MAX(CASE WHEN telephone_number_types.type = 'PREF' THEN 1 ELSE 0 END) AS pref,
 MAX(CASE WHEN telephone_number_types.type = 'HOME' THEN 1 ELSE 0 END) AS home,
 MAX(CASE WHEN telephone_number_types.type = 'WORK' THEN 1 ELSE 0 END) AS work,
 MAX(CASE WHEN telephone_number_types.type = 'VOICE' THEN 1 ELSE 0 END) AS voice,
 MAX(CASE WHEN telephone_number_types.type = 'FAX' THEN 1 ELSE 0 END) AS fax,
 MAX(CASE WHEN telephone_number_types.type = 'MSG' THEN 1 ELSE 0 END) AS msg,
 MAX(CASE WHEN telephone_number_types.type = 'CELL' THEN 1 ELSE 0 END) AS cell,
 MAX(CASE WHEN telephone_number_types.type = 'PAGER' THEN 1 ELSE 0 END) AS pager,
 MAX(CASE WHEN telephone_number_types.type = 'BBS' THEN 1 ELSE 0 END) AS bbs,
 MAX(CASE WHEN telephone_number_types.type = 'MODEM' THEN 1 ELSE 0 END) AS modem,
 MAX(CASE WHEN telephone_number_types.type = 'CAR' THEN 1 ELSE 0 END) AS car,
 MAX(CASE WHEN telephone_number_types.type = 'ISDN' THEN 1 ELSE 0 END) AS isdn,
 MAX(CASE WHEN telephone_number_types.type = 'VIDEO' THEN 1 ELSE 0 END) AS video
FROM telephone_numbers
LEFT JOIN telephone_numbers_types_rel ON telephone_numbers.telephone_number_id
     = telephone_numbers_types_rel.telephone_number_id
LEFT JOIN telephone_number_types ON telephone_numbers_types_rel.telephone_number_type_id
     = telephone_number_types.telephone_number_type_id
GROUP BY telephone_numbers.telephone_number_id
ORDER BY pref DESC, telephone_numbers.telephone_number_id DESC;

CREATE VIEW addresses_view AS
SELECT
 addresses.address_id,
 addresses.contact_id,
 addresses.post_office_box,
 addresses.extended_address,
 addresses.street_address,
 addresses.locality,
 addresses.region,
 addresses.postal_code,
 addresses.country_name,
 MAX(CASE WHEN address_types.type = 'PREF' THEN 1 ELSE 0 END) AS pref,
 MAX(CASE WHEN address_types.type = 'HOME' THEN 1 ELSE 0 END) AS home,
 MAX(CASE WHEN address_types.type = 'WORK' THEN 1 ELSE 0 END) AS work,
 MAX(CASE WHEN address_types.type = 'DOM' THEN 1 ELSE 0 END) AS dom,
 MAX(CASE WHEN address_types.type = 'INTL' THEN 1 ELSE 0 END) AS intl,
 MAX(CASE WHEN address_types.type = 'POSTAL' THEN 1 ELSE 0 END) AS postal,
 MAX(CASE WHEN address_types.type = 'PARCEL' THEN 1 ELSE 0 END) AS parcel
FROM addresses
LEFT JOIN addresses_types_rel ON addresses.address_id
     = addresses_types_rel.address_id
LEFT JOIN address_types ON addresses_types_rel.address_type_id
     = address_types.address_type_id
GROUP BY addresses.address_id
ORDER BY pref DESC, addresses.address_id DESC;
```

```
CREATE VIEW contacts_view AS
SELECT
 contacts.contact_id,
 contacts.honorific_prefixes AS title,
 contacts.family_name AS last_name,
 contacts.given_name AS first_name,
 contacts.bday AS birthday,
 NULL AS anniversary,
 contacts.org AS company,
 contacts.title AS job_title,
 home_phone_1.number as home_phone,
 home_phone_2.number as home_phone_2,
 work_phone_1.number as work_phone,
 work_phone_2.number as work_phone_2,
 mobile_phone.number as mobile_phone,
 pager_phone.number as pager_phone,
 fax_phone.number as fax_phone,
 other_phone.number as other_phone,
 email_1.email AS email_1,
 email_2.email AS email_2,
 email_3.email AS email_3,
 home_address.street_address AS home_address_1,
 home_address.extended_address AS home_address_2,
 home_address.locality AS home_address_city,
 home_address.country_name AS home_address_country,
 home_address.region AS home_address_state_province,
 home_address.postal_code AS home_address_zip_postal,
 work_address.street_address AS work_address_1,
 work_address.extended_address AS work_address_2,
 work_address.locality AS work_address_city,
 work_address.country_name AS work_address_country,
 work_address.region AS work_address_state_province,
 work_address.postal_code AS work_address_zip_postal,
 photos.photo_uri AS picture,
 NULL AS pin,
 contacts.uid AS uid,
 contacts.url AS web_page,
 (SELECT GROUP_CONCAT(categories.category) FROM categories
        WHERE categories.contact_id = contacts.contact_id) AS categories,
 contacts.note AS note,
 NULL AS user1,
 NULL AS user2,
 NULL AS user3,
 NULL AS user4
from contacts
LEFT JOIN telephone_numbers_view home_phone_1 ON contacts.contact_id
     = home_phone_1.contact_id AND home_phone_1.home = 1
LEFT JOIN telephone_numbers_view home_phone_2 ON contacts.contact_id
     = home_phone_2.contact_id AND home_phone_2.home = 1
 AND home_phone_2.telephone_number_id <> home_phone_1.telephone_number_id
LEFT JOIN telephone_numbers_view work_phone_1 ON contacts.contact_id
     = work_phone_1.contact_id AND work_phone_1.work= 1
LEFT JOIN telephone_numbers_view work_phone_2 ON contacts.contact_id
     = work_phone_2.contact_id AND work_phone_2.work = 1
 AND work_phone_2.telephone_number_id <> work_phone_1.telephone_number_id
LEFT JOIN telephone_numbers_view mobile_phone ON contacts.contact_id
     = mobile_phone.contact_id AND mobile_phone.cell = 1
LEFT JOIN telephone_numbers_view pager_phone ON contacts.contact_id
     = pager_phone.contact_id AND pager_phone.pager = 1
LEFT JOIN telephone_numbers_view fax_phone ON contacts.contact_id
     = fax_phone.contact_id AND fax_phone.fax = 1
LEFT JOIN telephone_numbers_view other_phone ON contacts.contact_id
     = other_phone.contact_id
 AND other_phone.telephone_number_id NOT IN(
        COALESCE(home_phone_1.telephone_number_id, 0),
  COALESCE(home_phone_2.telephone_number_id, 0),
```

```
  COALESCE(work_phone_1.telephone_number_id, 0),
  COALESCE(work_phone_2.telephone_number_id, 0),
  COALESCE(mobile_phone.telephone_number_id, 0),
  COALESCE(pager_phone.telephone_number_id, 0),
  COALESCE(fax_phone.telephone_number_id, 0))
LEFT JOIN emails_view email_1 ON contacts.contact_id = email_1.contact_id
LEFT JOIN emails_view email_2 ON contacts.contact_id = email_2.contact_id
 AND email_2.email_id <> email_1.email_id
LEFT JOIN emails_view email_3 ON contacts.contact_id = email_3.contact_id
 AND email_3.email_id <> email_1.email_id AND email_3.email_id <> email_2.email_id
LEFT JOIN addresses_view home_address ON contacts.contact_id
     = home_address.contact_id AND home_address.home = 1
LEFT JOIN addresses_view work_address ON contacts.contact_id
     = work_address.contact_id AND work_address.work = 1
LEFT JOIN photos ON contacts.contact_id = photos.contact_id
WHERE 0=0
AND (CASE WHEN home_phone_2.telephone_number_id IS NOT NULL
        THEN home_phone_1.pref >= home_phone_2.pref ELSE 1 END)
AND (CASE WHEN work_phone_2.telephone_number_id IS NOT NULL
        THEN work_phone_1.pref >= work_phone_2.pref ELSE 1 END)
AND (CASE WHEN email_2.email_id IS NOT NULL THEN email_1.pref >= email_2.pref ELSE 1 END)
AND (CASE WHEN email_3.email_id IS NOT NULL THEN email_1.pref >= email_3.pref ELSE 1 END)
GROUP BY contacts.contact_id
ORDER BY LOWER(last_name) ASC, LOWER(first_name) ASC;
```

Messages database

The messages database contains all the MAP-specific data.

ER diagram

The following entity-relationship (ER) diagram shows the relationships among the tables in the messages database:

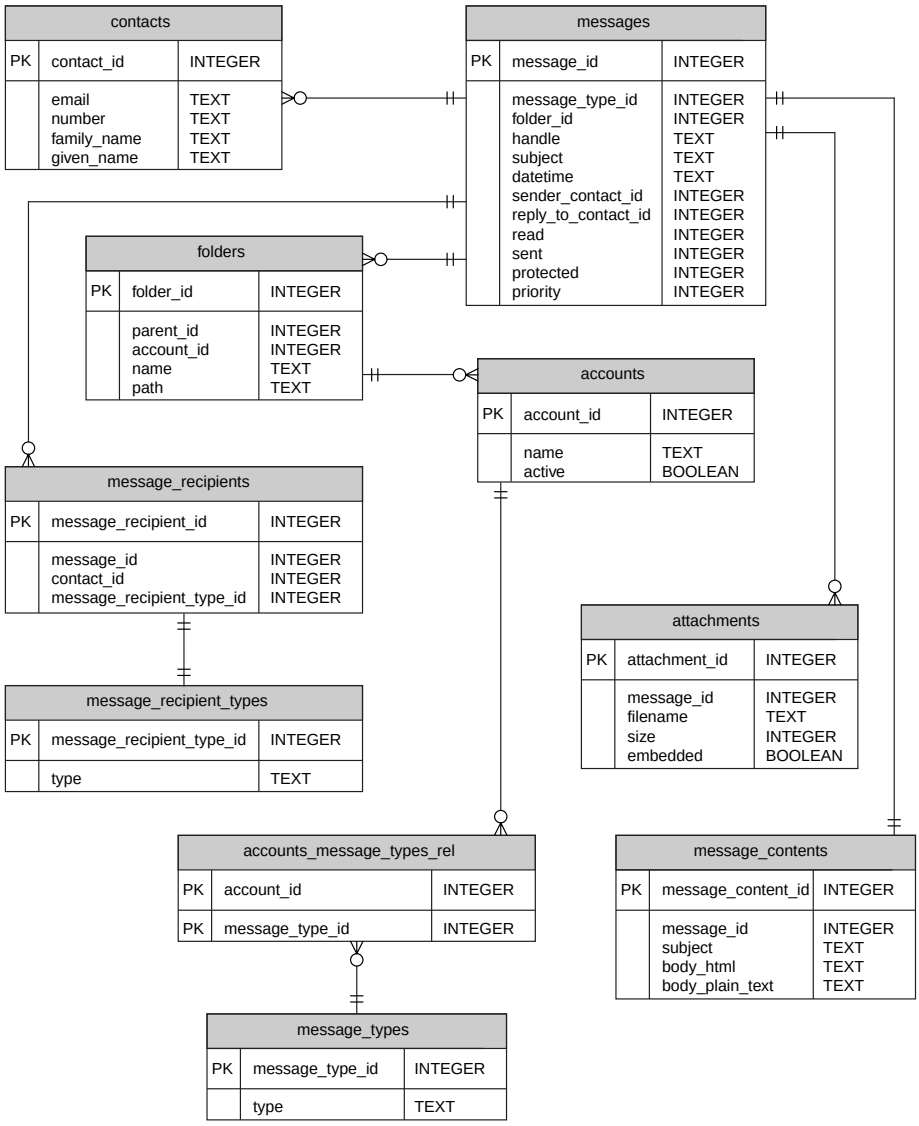

Figure 8: ER diagram for messages database

Sample schema file

The database schema is specified in `/db/messages.sql`, which looks like this:

```
/* Turn foreign key constraints on */
```

```
PRAGMA foreign_keys = ON;

/* Message types */
CREATE TABLE message_types (
 message_type_id  INTEGER PRIMARY KEY AUTOINCREMENT,
 type    TEXT NOT NULL UNIQUE
);

/* Accounts/Instances */
CREATE TABLE accounts (
 account_id INTEGER PRIMARY KEY,
 name  TEXT NOT NULL UNIQUE,
 active  BOOLEAN NOT NULL
);

/* Accounts/Instances */
CREATE TABLE accounts_message_types_rel (
 account_id   INTEGER NOT NULL REFERENCES accounts ON DELETE CASCADE,
 message_type_id  INTEGER NOT NULL REFERENCES message_types,
 PRIMARY KEY (account_id, message_type_id)
);

/* Folders */
CREATE TABLE folders (
 folder_id INTEGER PRIMARY KEY AUTOINCREMENT,
 parent_id INTEGER REFERENCES folders ON DELETE CASCADE,
 account_id INTEGER NOT NULL REFERENCES accounts ON DELETE CASCADE,
 name  TEXT NOT NULL,
 path  TEXT NOT NULL,
 CHECK (parent_id <> folder_id)
);

/* Messages (all types: email, sms, mms) */
CREATE TABLE messages (
 message_id   INTEGER PRIMARY KEY AUTOINCREMENT,
 message_type_id  INTEGER NOT NULL REFERENCES message_types,
 folder_id   INTEGER NOT NULL REFERENCES folders ON DELETE CASCADE,
 handle    TEXT NOT NULL,
 subject    TEXT NOT NULL,
 datetime   TEXT NOT NULL,
 sender_contact_id INTEGER NOT NULL REFERENCES contacts(contact_id),
 reply_to_contact_id INTEGER REFERENCES contacts(contact_id),
 read    INTEGER NOT NULL DEFAULT 0,
 sent    INTEGER NOT NULL DEFAULT 0,
 protected   INTEGER NOT NULL DEFAULT 0,
 priority   INTEGER NOT NULL DEFAULT 0,
 CHECK     (read = 0 OR read = 1),
 CHECK     (sent = 0 OR sent = 1),
 CHECK     (protected = 0 OR protected = 1),
 CHECK     (priority = 0 OR priority = 1)
);

/* Contacts (senders/recipients) */
CREATE TABLE contacts (
 contact_id INTEGER PRIMARY KEY AUTOINCREMENT,
 email  TEXT,
 number  TEXT,
 family_name TEXT,
 given_name TEXT,
 CHECK(email IS NOT NULL OR number IS NOT NULL)
);

/* Message contents */
CREATE TABLE message_contents (
 message_content_id INTEGER PRIMARY KEY AUTOINCREMENT,
 message_id   INTEGER NOT NULL REFERENCES messages ON DELETE CASCADE,
 subject    TEXT ,
```

```
 body_html   TEXT ,
 body_plain_text  TEXT ,
 CHECK    (body_html IS NOT NULL OR body_plain_text IS NOT NULL)
);

/* Message recipient types */
CREATE TABLE message_recipient_types (
 message_recipient_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
 type      TEXT NOT NULL UNIQUE
);

/* Message recipients */
CREATE TABLE message_recipients (
 message_recipient_id  INTEGER PRIMARY KEY AUTOINCREMENT,
 message_id     INTEGER NOT NULL REFERENCES messages ON DELETE CASCADE,
 contact_id     INTEGER NOT NULL REFERENCES contacts,
 message_recipient_type_id INTEGER NOT NULL REFERENCES message_recipient_types
);

/* Attachments */
CREATE TABLE attachments (
 attachment_id INTEGER PRIMARY KEY AUTOINCREMENT,
 message_id  INTEGER NOT NULL REFERENCES messages ON DELETE CASCADE,
 filename  TEXT NOT NULL,
 size   INTEGER NOT NULL,
 embedded  BOOLEAN NOT NULL
);

/**
 VIEWS
 */

 /* View to retrieve a list of brief messages */
 CREATE VIEW "messages_view" AS
 SELECT
 accounts.account_id,
 accounts.name as account_name,
 messages.message_id,
 messages.folder_id,
 folders.name as folder_name,
 folders.path as folder_path,
 message_types.type,
 messages.handle,
 messages.subject,
 messages.datetime,
 messages.sender_contact_id,
 contacts_sender.email as sender_email,
 contacts_sender.number as sender_number,
 contacts_sender.family_name as sender_last_name,
 contacts_sender.given_name as sender_first_name,
 messages.reply_to_contact_id,
 contacts_reply.email as reply_to_email,
 contacts_reply.number as reply_to_number,
 contacts_reply.family_name as reply_to_last_name,
 contacts_reply.given_name as reply_to_first_name,
 messages.read,
 messages.sent,
 messages.protected,
 messages.priority,
 recipients.email as recipient_email,
 recipients.number as recipient_number,
 recipients.family_name as recipient_last_name,
 recipients.given_name as recipient_first_name
FROM messages
LEFT JOIN contacts contacts_sender ON messages.sender_contact_id
     = contacts_sender.contact_id
LEFT JOIN contacts contacts_reply ON messages.reply_to_contact_id
```

```
     =  contacts_reply.contact_id
LEFT JOIN folders ON messages.folder_id =  folders.folder_id
LEFT JOIN accounts ON folders.account_id = accounts.account_id
LEFT JOIN message_types ON messages.message_type_id
     = message_types.message_type_id
LEFT JOIN contacts recipients ON recipients.contact_id
     = (SELECT contact_id FROM message_recipients
        WHERE message_recipients.message_id = messages.message_id
        AND message_recipients.message_recipient_type_id
         = 1 ORDER BY message_recipients.message_recipient_id DESC LIMIT 1);

/* View to retrieve full messages */
CREATE VIEW "full_messages_view" AS
SELECT
 accounts.account_id,
 accounts.name as account_name,
 messages.message_id,
 messages.folder_id,
 folders.name as folder_name,
 folders.path as folder_path,
 message_types.type,
 messages.handle,
 messages.datetime,
 messages.sender_contact_id,
 contacts_sender.email as sender_email,
 contacts_sender.number as sender_number,
 contacts_sender.family_name as sender_last_name,
 contacts_sender.given_name as sender_first_name,
 messages.reply_to_contact_id,
 contacts_reply.email as reply_to_email,
 contacts_reply.number as reply_to_number,
 contacts_reply.family_name as reply_to_last_name,
 contacts_reply.given_name as reply_to_first_name,
 messages.read,
 messages.sent,
 messages.protected,
 messages.priority,
 COALESCE(message_contents.subject, messages.subject) as subject,
 message_contents.body_plain_text,
 message_contents.body_html
FROM messages
LEFT JOIN message_contents ON messages.message_id
     = message_contents.message_id
LEFT JOIN contacts contacts_sender ON messages.sender_contact_id
     = contacts_sender.contact_id
LEFT JOIN contacts contacts_reply ON messages.reply_to_contact_id
     = contacts_reply.contact_id
LEFT JOIN folders ON messages.folder_id =  folders.folder_id
LEFT JOIN accounts ON folders.account_id = accounts.account_id
LEFT JOIN message_types ON messages.message_type_id
     = message_types.message_type_id
WHERE message_contents.message_content_id IS NOT NULL;

/* view to retrieve contacts */
CREATE VIEW "contacts_view" AS
select contacts.contact_id, contacts.email, contacts.number, contacts.family_name,
    contacts.given_name, message_recipient_types.type,  message_recipients.message_id
     from message_recipients
LEFT JOIN message_recipient_types ON message_recipients.message_recipient_type_id
     = message_recipient_types. message_recipient_type_id
LEFT JOIN contacts ON message_recipients.contact_id =  contacts.contact_id;


/*
 TRIGGERS
*/
```

```
/* Constrain messages to be a message type that is of its parent account supported
 * message types */
/*
CREATE TRIGGER insert_message_check_message_type BEFORE INSERT ON messages
FOR EACH ROW WHEN NOT EXISTS (SELECT *
  FROM accounts_message_types_rel
  LEFT JOIN folders ON accounts_message_types_rel.account_id = folders.account_id
  LEFT JOIN messages ON folders.folder_id = new.folder_id
            WHERE new.message_type_id = accounts_message_types_rel.message_type_id)
BEGIN
 SELECT RAISE(ABORT,
              'Message type must be a supported message type of the message''s account');
END

CREATE TRIGGER insert_message_check_handle BEFORE INSERT ON messages
FOR EACH ROW WHEN (SELECT count(*) FROM messages JOIN folders
    ON messages.folder_id = folders.folder_id
    WHERE new.handle = messages.handle AND folders.account_id =
        (SELECT accounts.account_id FROM accounts JOIN folders
         ON accounts.account_id = folders.account_id
        WHERE folders.folder_id = new.folder_id)) > 0
BEGIN
 SELECT RAISE(ABORT, 'Handle must be unique per a message''s account');
END
*/
```

Index

S

T

W