Automatic Speech Recognition Developer's 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: Thursday, February 20, 2014

Table of Contents

About This Guide

This guide describes the Automatic Speech Recognition (ASR) subsystem. The ASR subsystem provides end-to-end handling of spoken commands, utilizing a module-based, extensible architecture.

This guide is intended for developers who will be modifying and extending the ASR subsystem of the QNX CAR platform.

The following table may help you find information quickly:

To find out about:	Go to:
The architecture and process control flow of the ASR subsystem	*Automatic Speech Recognition* (p. 9)
How to extend the ASR subsystem	*Extending ASR* (p. 13)
Common parts of a module	*Anatomy of a module* (p. 14)
Details on the conversation modules	*Conversation modules* (p. 19)
How to add a conversation module	*Adding a conversation module* (p. 31)
Functions, data types, structures, etc.	*API Reference* (p. 35)

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
Automatic Speech Recognition

The ASR subsystem (`io-asr`) offers complete speech-recognition services and supports Nuance VoCon Hybrid 4.4 for speech-to-text (STT) and text-to-speech (TTS) conversion. In addition, a reference implementation using AT&T Watson can help guide you in integrating the ASR subsystem with other recognition providers.

To use VoCon, you must have a separate licensing agreement and NDA with Nuance.

ASR provides speech-recognition services in the following areas:

- search—launch an application, get weather information, get directions or search for points of interest
- multimedia—play tracks by artist, album, genre, or song title, and control playback (pause, previous track, next track, and so on).
- voice dialing—by contact name or by number (requires a connected Bluetooth device)

The `io-asr` service

The ASR service is referred to as `io-asr` throughout this guide. However, `io-asr` is a shorthand term for the actual service that runs. The name of the service depends on which recognizer is being used. The actual service is `io-asr-`*recognizer_name*, for example, `io-asr-vocon`.

Modules

ASR uses *modules* to perform the various functions that provide end-to-end handling of spoken commands. It doesn't handle speech by itself; it passes information between modules so that they can perform the various stages of recognizing and taking action on spoken commands. The modules interact with `io-asr`, not directly with each other.

The ASR subsystem has four types of modules:

- **prompt**—requests information from the user. Prompts can be audio (spoken directions or another sound such as a beep or a bell) or visual (for example, text written to a portion of the display, a prompt screen with various options, or some other visual cue such as a change of icon or color).
- **audio**—listens for commands and captures audio from the microphone (capture module), or plays audio back to the user (file module).

- **recognition**—converts speech to text. After the audio module has captured a command (called an "utterance"), ASR instructs the recognition module to convert the captured audio command to a text string that can be passed to a converstaion module.
- **conversation**—interprets the text command and takes the appropriate action. There are conversation modules for three types of commands: search, multimedia, voice dialing. Each of these conversation modules has its own configuration that determines the grammar used to interpret the command. Once a command is successfully interpreted, the conversation module invokes the subsystem required and passes it the information it needs to complete the request. Often the required action is invoking another ASR pass to get more information from the user.

ASR startup

When the target system boots, System Launch and Monitor (SLM) starts ASR and passes it the path to the configuration file (`${QNX_TARGET}/etc/asr-car.cfg`). ASR begins to run as a daemon, reads the configuration file, and loads it into memory. It then loads the modules as specified by the configuration file by calling *dlopen()* for the associated DLLs. Each module has a constructor function that registers it with ASR by calling the module's connect function. The module's constructor function calls one of the following, depending on what type of module it is:

- *asra_connect()* (p. 56) for an audio module
- *asrm_connect()* (p. 68) for a conversation module
- *asr_connect()* (p. 186) for a recognition module
- *asrp_connect()* (p. 114) for a prompt module

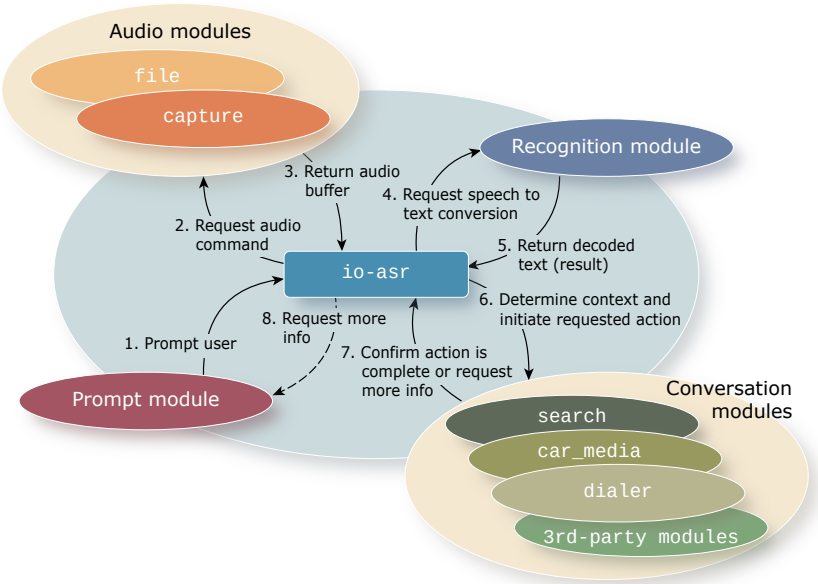
ASR then invokes each registered module's initialize callback function, which initializes any private or module-specific data. At this point, ASR is ready to handle voice commands.

PPS integration

ASR uses PPS to communicate with the HMI. See `/pps/services/asr/control` in the *PPS Objects Reference* for more information.

Process control flow

ASR operates in a cyclical fashion, performing the same sequence of operations as often as required to complete the user's request. Each cycle of these operations is referred to as a *recognition turn* and is illustrated in the following diagram:



ASR's control flow works as follows:

1. ASR is triggered by the prompt module, which monitors the system for events (a UI button press or a PPS update, for example) and then starts a recognition turn by prompting the user for a command.
2. After the prompt is rendered, ASR passes control to the audio capture module to capture the user's spoken command. On a successful capture, control passes to the recognition module. If the command capture isn't successful, control passes back to the prompt module to retry.
3. The recognition module converts the audio command to a text string and assigns the result a level of confidence to indicate how well the command was "understood" by the recognizer. Depending on the configuration, if the confidence level isn't high enough, ASR will prompt the user again.
4. When a successful result is available, ASR passes control to the conversation modules. The conversation modules must first determine the context of the command (e.g., search, multimedia, or phone). The context determines which conversation module takes over to complete the command. When a context is determined, the associated conversation module is "exclusive." That is, it's the only conversation module that will handle results until this command is fulfilled. At this point, the exclusive module either completes the action or triggers another recognition turn to have the user prompted again for more information. This process

continues until the action is completed. The conversation module then removes its exclusive status so that a fresh recognition turn can proceed.

Chapter 2
Extending ASR

ASR offers complete voice-control services for search, media, and voice dialing, but there may be circumstances where you want to modify functionality in existing modules or add additional modules. The most common module to extend or add is the conversation module. This allows additional functionality to be triggered by voice commands without changing the way the ASR subsystem operates.

If you want to use a different recognizer, you'll need to reimplement the recognizer modules. You might also need to modify or reimplement the prompt modules. See the AT&T Watson reference implementation for guidance or contact QNX technical support.

Anatomy of a module

While the internal implementation of the modules is quite different, they all share common features. Each type of module has an interface that defines how it interacts with `io-asr`:

- `asr_recognizer_if_t` for the recognizer module
- `asrp_module_interface_t` (p. 107) for the prompt module
- `asra_module_interface_t` (p. 185) for the audio modules
- `asr_conversation_if_t` (p. 171) for the conversation modules

Each interface includes some or all of the following members:

- *name* (all modules)—the name of the module
- *version* (all modules)—the version of the of `libasr-core` that the module was built against. The `io-asr` service checks this version number to ensure that the module is compatible with the current system.
- *init()* (all modules)—a callback function that performs whatever initialization tasks are required to get the module ready to handle requests from `io-asr`
- *start()* (prompt, audio, recognizer)—a callback function that causes the module to begin performing its particular operation
- *stop()* (all modules)—a callback function that causes the module to stop performing its particular operation
- *step()* (prompt, recognizer), *on_asr_step()* (conversation)— callback functions that perform actions depending on the current step (`asr_step_t`) of the recognizer

All modules provide an initialization function that `io-asr` invokes when the module is loaded. In addition to setting up module-specific data, the initialization function also loads the configuration tree and extracts the confidence threshold that's used to select speech-to-text results from the recognizer. The confidence threshold value is set globally per instance of the `io-asr` subsystem. This threshold is used frequently; saving it to memory speeds up access since the module won't have to walk the configuration tree each time the threshold is required.

Many of the callback functions in the module interfaces take private module-specific data as an argument (*init()*, *destroy()*, *start()*, *stop()*). This argument is a void pointer that's usually named *module_data*. ASR doesn't access this data, so its structure can be whatever is required by the individual module; two modules of the same type might use different module-specific data. This module-specific data structure can be passed to `io-asr` via the module's *connect()* (p. 10) function. ASR can then pass it as an argument to the module's callback functions.

Prompt module

The prompt module is responsible for triggering ASR. It has a separate thread that monitors the PPS control object for changes and then triggers a recognition turn based on those changes. On an initial prompt (a new recognition turn), the prompt module plays a voice prompt that asks the user for a command. On subsequent turns within the same session, the prompts may take different forms depending on what's required. For example, if the command wasn't understood, a voice prompt asks for the command to be repeated. For media playback, there may be a spoken confirmation as well as a visual change to the HMI to indicate that playback is starting.

The prompts that are rendered for various situtuations are defined by the prompt configuration files, `*prompts.cfg`. These files reside at various locations in the ASR subsystem depending on their purpose.

The prompts to be rendered for the various conversation modules are defined in `car-prompts.cfg` or `prompts.cfg` files in the `${`***`QNX_TARGET`***`}/opt/asr/conversation/`*`locale/module`* directory. For example, the English prompts for media playback are in the `${`***`QNX_TARGET`***`}/opt/asr/conversation/en-US/car-media` directory

The prompt module is tightly integrated with the HMI and the audio drivers. You should be able to adjust the prompt module to suit your needs just by modifying the configuration files. If you need more extensive changes, please contact QNX technical support.

Audio modules

There are two audio modules:

- `capture`, which acquires the user's spoken commands
- `file`, which plays back audio files (usually verbal prompts from the ASR system)

The `capture` audio module acquires audio from the microphone via `libasound`. The captured audio is placed in a buffer that `io-asr` passes to the recognition module to be converted to text.

The `file` audio module plays back audio from a WAV file. The audio module sets up the required audio configuration, and then passes the audio file to the prompt module. The prompt module invokes Audio Manager to perform the actual playback of the file.

The audio modules are tightly integrated with the audio drivers, Audio Manager, and `libasound`. You shouldn't modify the audio modules. If you require changes, please contact QNX technical support.

Recognition module

The recognition module converts a spoken command (i.e., an utterance) to text. It collects the audio sample, passes it to the vendor's text-to-speech service for processing, and converts the vendor-specific result data to the format required by ASR. The recognition module returns a *dictation* result, which is strictly a speech-to-text conversion. The result is returned in the `asr_result` (p. 205) type).

The `io-asr` service passes the dictation result to the Natural Language Adaptation Layer (NLAL), which extracts *intent* information (`asr_intent` (p. 204)). This intent data is added to the original result structure. The NLAL uses BNF grammars provided with the conversation modules to analyze the dictation result to extract its meaning and produce intent results.

In some cases, the vendor's text-to-speech service can extract salient information from the utterance to construct an intent result. However, the dictation result must always be included so that ASR can perform its own natural language processing on the utterance.

For example, the utterance "search media for Hero" could result in the following dictation result from the recognizer:

- result type: Dictation
- utterance: "search media for Hero"
- start rule: `search#media-search`
- confidence: *xxx* (a value in the range 0-1000, where 1000 is 100% confidence)

The NLAL would then analyze this dictation result to create the following intent result:

- result type: Intent
- utterance: "search media for Hero"
- start rule: `media-search`
- confidence: *xxx*
- intent entries: 2
    - field: "search-type", value: "media"
    - field: "search-term", value: "Hero"

The final recognition result (with intent information) is passed to the conversation modules to be interpreted and acted upon. The intent fields are vital for the conversation modules—the conversation can't take place if the system can't understand the meaning of an utterance.

The recognition module is tightly integrated with the third-party ASR vendor. If you require changes or want to use a different vendor, see the AT&T Watson reference application for guidance or contact QNX technical support.

Conversation modules

The conversation modules interpret the recognized speech result and take the appropriate action. There are many conversation modules; each handles a specific type of result. When a new intent result is available, `io-asr` passes it to all the conversation modules to provide a rating.

ASR selects the module that provides the highest rating and passes it the current result for processing. The module can optionally set itself as exclusive by calling the *asrm_set_exclusive()* (p. 92) function so that it will receive all future results in the current speech session (i.e., `io-asr` won't evaluate the ratings of other conversation modules). The module releases its exclusive status when ASR transitions to the `ASR_STEP_SESSION_CLOSED` state.

Each module has one or more configuration files that describe the grammar it uses to interpret the recognition result, and then maps the outcomes to specific actions (for example, play a named track, dial a contact phone number, and so on). The modules load these configuration files into memory as configuration trees (`cfg_item_t` (p. 134)).

The configuration files are found in the `asr/conversation/`*`locale/context`* directory. For example, the English configuration files for the `search` module are found in the `asr/conversation/en-US/search` directory.

All conversation modules must implement the *select_result()* callback function of the conversation module interface, `asr_conversation_if` (p. 168). This callback evaluates the results obtained from the recognizer and selects the result that's most relevant. It assigns a score to the result, allowing `io-asr` to determine which module should be used to handle the result. This function requires three arguments:

- *result_set*—the set of results that were obtained for a given speech utterance. The utterance is translated into one or more tagged string sequences and collected within this argument.
- *module_data*—the private module data associated with the `car-media` module.
- *selected_result*—a pointer to a buffer that contains the address of the selected result (from the results set). This argument is updated only in the event that the module finds a relevant result in the provided result set.

If a relevant result is found, the function assigns it to the *selected_result* argument and then returns the module's confidence that the selected result is relevant. If a competing module matches a result with a higher confidence, `io-asr` selects that module rather than the current one.

If this function doesn't select a result, it returns a confidence score of zero.

It's possible that the command uttered by the user only approximately matches a known command. For this reason, ASR assigns a confidence score to each of the pending actions. This allows the module to request clarification from the user when the uncertainty of the command is too high (i.e., the confidence score is less than the confidence threshold). In this case, the module ignores the pending action and `io-asr` prompts the user to repeat their last command.

Instead of using the globally set confidence threshold to determine the relevance of a result, the conversation module may optionally use a per-module confidence threshold.

The conversation modules can output log information to the system logger at varying verbosity levels. The modules use the *asrm_slog()* (p. 95) function to output log information, allowing the verbosity to be set via the ASR configuration file. In addition, if you compile a module with the DEBUG preprocessor symbol defined, the logging information will include the filename, line number, and function name for all messages issued by the module.

The `search` module

The `search` module allows the user to perform various searches and to launch other applications.

The `search` module, like all other modules, provides an initialization callback function. After `io-asr` has loaded all modules, it invokes the initialization callback to set up the private data structures managed by the module.

The initialization callback loads the `search`-specific configuration tree to make it available to the module. This configuration data is assumed to be in the `module/search` path of the global configuration tree. The `search` module has two locale-specific configuration files that are used to customize its behavior:

- `car-control.cfg`—defines text keys that are used to tag terminals in a recognition result. You can add more keys to this section to trigger search commands with custom keywords.
- `car-prompts.cfg`—defines the locale-specific prompts that are used by the `search` module.

When `io-asr` shuts down, it releases all modules that it has dynamically loaded. The `search` module provides the *destroy()* callback, which releases resources allocated during the initialization phase before the module is deleted from the ASR subsystem.

Actions

The `search` conversation module can:

- query the time
- query the date

- launch or close an application
- get the current weather in a specified location
- get directions to a point of interest (POI) or address
- search for media or a POI

In addition to these actions, the `search` module also provides a general help prompt that tells the user what voice commands are supported.

Conversation flow

The `search` module can handle a number of different states that update the result action (`result_action_e` (p. 216)). If a search command is successful, the result action is set to `ASR_RECOGNITION_COMPLETE`. If insufficient information was provided in the utterance, the result action is set to `ASR_RECOGNITION_RESTART` to reprompt the user. The `search` module's states are as follows:

- `IDLE`—the default state of the `search` module. From this state, the module can issue search instructions. There is no change to the result action.
- `MEDIA_SEARCH`, `RADIO`, `AUDIO`, and `VIDEO`—invoke the `mm-player` service to complete the action.
- `SEARCH`—an intermediate state in the process of handling a media or POI search (while the domain of the search is being determined).
- `POI` and `NAV`—invoke the navigation service to get directions.
- `WEATHER`—invoke the weather application to get the current weather for a location.
- `HELP`—play the help prompt to the user.
- `CANCEL`—abort the last spoken command. The result action is set to `ASR_RECOGNITION_CANCEL`.
- `LAUNCH`—launch the specified application. The `search` module maintains a mutex-protected list of applications that it keeps current.
- `CLOSE`—close the specified application.
- `TIME`—get the time of day from the system.
- `DATE`—get the date from the system.

ASR state transitions

The `search` module keeps track of the current ASR state. This state affects what actions are taken by the module when a callback function is invoked (the result handler in particular). To keep track of the current ASR step, the `search` module defines a *step()* callback function that `io-asr` invokes to inform the module of the current state in the speech-recognition process. This function takes the current step ID and a pointer to the module data as arguments.

The *step()* function handles the following transitions:

- ASR_STEP_LOCALE_CHANGED—the configuration of the module may need to change. The *step()* function ensures that the correct locale configuration information is associated with the module's data. Localized data includes grammar configuration and prompt data.
- ASR_STEP_SESSION_OPENED—places the `search` module into the `IDLE` state and makes these grammar configuration sections active: base, base intents, and common.
- ASR_STEP_SESSION_CLOSED—removes the `search` module's status as the exclusive recipient of results.
- ASR_STEP_RECOGNITION_BEGIN—ensures that the correct configuration section is set as active when a speech-recognition session is started. If the module is not in the idle state, the confirmation section is set as active; otherwise, the base section is set as active.
- ASR_STEP_PRE_AUDIO_CAPTURE—places the module into the idle state.
- ASR_STEP_RECOGNITION_END—unlocks the application list mutex.

The *step()* function ignores all other state transitions.

Result handling

The *on_result()* callback function of the conversation module interface, `asr_conversation_if` (p. 168), processes speech-to-text results that the *select_result()* callback function has determined are relevant to the current module. The *on_result()* function is responsible for handling the selected result according to the conversation flow, as described earlier in this section. If the speech-to-text result doesn't provide enough information to complete the action, the *on_result()* function forces a restart of the speech-recognition routine to obtain further information.

Grammars

The `search` module provides two types of grammars that define its behaviour. One type of grammar affects how the intents of an utterance are extracted from a speech-to-text result. The other type is supplied to a third-party recognizer to build a context for performing the actual speech-to-text conversion.

The mapping from speech to `search` actions is defined by a grammar that is specified in a localized configuration file. This file is loaded by the ASR subsystem at execution time and parsed into a configuration tree. The subtree that is relevant to the `search` module is then copied to its private data structure.

The `search` module also provides localized BNF grammars that are used to compile recognizer contexts that control speech-to-text transformations. These grammars describe the key phrases that can be used to interact with the `search` module and that provide a list of applications that are present on the system. The default context grammars are defined in `<recognizer>/en-US/search/car-search.bnf` and `<recognizer>/en-US/search/car-applications.bnf`.

The `car-media` module

The `car-media` module uses one of two multimedia services to perform media operations: `mm-player` or `mm-control`. From the user's perspective, these services behave exactly the same, but what happens behind the scenes is different, so there are separate plugins. Each of the plugins provides different private data, aggregated into the main `car-media` data structure. These distinct plugins can be loaded exclusively to provide multimedia voice support, but they both can't be loaded simultaneously.

Each media plugin of the `car-media` module provides an initialization function, which is invoked when the module is registered with the ASR system. Similarly, each plugin provides a tear-down function that ensures that any allocated resources are released when the module is unloaded. This function is called by the *destroy()* callback function.

The `mm-player` plugin

The `mm-player` back end of the `car-media` module relies on a C API to communicate with the media service. This requires the plugin to open a handle to the `mm-player` service. For more information about the `mm-player` API, see the *Multimedia Player Developer's* Guide

The `mm-control` plugin

The `mm-control` plugin of the `car-media` module communicates with the media service using the `control` and `status` PPS objects for `mm-control`.

The `mm-control` service is being deprecated in favour of `mm-player`.

The `control` object provides an access point for clients to publish commands and messages to the `mm-control` service. The `mm-control` service publishes status information about the media, such as the ID of current track and the playback speed, to the `status` object.

The `car-media` module defines a helper function that parses the data from the PPS objects.

For more information about the `mm-control` service, see the *Multimedia Controller Configuration Guide*. For information specific to the `mm-control` PPS objects, see the `/pps/services/mm-control/control` and `/pps/services/mm-control/<playername>/status` entries of the *PPS Objects Reference*.

Actions

The `car-media` module defines a number of actions that can be initiated via voice control. These are grouped into high-level workflows that aren't directly tied to specific commands words or utterances, making the module easily adaptable to multilingual environments. Each action ID is associated with a rule string. To enable the `car-media` module for a particular language environment, a locale-specific grammar simply associates a grammar entry to the rule string corresponding to the correct action ID.

The actions supported by the `car-media` modules are:

- Media playback control—allows the playback of media to be controlled by voice. The ID and rule string associated with this particular action are as follows:

  ```
  [ MEDIA_CTRL ]          = "media-ctrl",
  ```

  The `media-ctrl` action is further subdivided into actions that express specific controls being executed by the module as follows:

  ```
  [ PAUSE ]               = 0, /* media-ctrl */
  [ RESUME ]              = 0, /* media-ctrl */
  [ PREVIOUS ]            = 0, /* media-ctrl */
  [ NEXT ]                = 0, /* media-ctrl */
  ```

- Media search and tracksession generation—allows the user to create a new track session (populated with media files matching a particular search string and type) and to start playing the tracks in order. The ID and rule string associated with this particular action are as follows:

  ```
  [ PLAY ]                = "media-playback"
  ```

  The media search action is further subdivided into actions that narrow the scope of the search into categories:

  ```
  [ ALBUM_PLAYBACK ]      = "album-playback"
  [ SONG_PLAYBACK ]       = "song-playback"
  [ ARTIST_PLAYBACK ]     = "artist-playback"
  ```

Conversation flow

The action IDs described previously are used internally by the module to specify the current state of the conversation flow. Two states must be accounted for:

- Current—the state following the `car-media` module's turn in the conversation. In other words, this is the state immediately after the module has responded to the user (or more accurately, produced prompt information for one of the prompt modules). Initially, the module's conversation state is IDLE.
- Pending—the state of the conversation after the user's utterance is considered. The utterance constitutes a command that may need to be executed by the module. When the command is executed, the pending state becomes the current state.

ASR state transitions

Like the `search` module, the car-media model keeps track of the current ASR state. For details, see *ASR state transitions* (p. 21) in the section "The `search` module".

The *step()* function handles the following transitions:

- `ASR_STEP_LOCALE_CHANGED`—the configuration of the module may need to change. The *step()* function ensures that the correct locale configuration information is associated with the module's data. Localized data includes grammar configuration and prompt data.
- `ASR_STEP_SESSION_OPENED`—places the `car-media` module into the `IDLE` state and makes these grammar configuration sections active: base, base intents, and common.
- `ASR_STEP_SESSION_CLOSED`—removes the `car-media` module's status as the exclusive recipient of results.

Result handling

Like the `search` module, the `car-media` module uses the *on_result()* callback function to process recognition results. For details, see *Result handling* (p. 22) in the section "The `search` module".

Grammars

Like the `search` module, the `car-media` module provides two grammars that define its behavior (one for extracting intents and one for building a context for the third-party recognizer). For details, see *Grammars* (p. 22) in the section "The `search` module".

The `car-media` grammar can be separated into two high-level groups: control and search. The control group involves all conversations that control the playback. The search group involves all conversations that set up tracksessions based on specified search terms and categories.

In addition, a default grammar for the media metadata list is provided. This grammar defines the *media-name* slot as being a null feature (i.e. it will not be matched). The context created by this grammar is replaced when new media is added to the system. This context simply serves as a placeholder, allowing the `car-media` module to work until a proper media data context is generated. This substitution is done at runtime as described in the following section.

Updating the guest context

The `car-media` module defines several helper functions that are used to construct an internal list of media data. The media metadata is obtained from internal databases that are created when media sources are connected to the device.

The media metadata is constructed in three passes:

1. song titles
2. artist names
3. album names

These three passes are repeated once for each synchronized media source. Synchronized media sources are listed by querying the `mm-detect` PPS `status` object.

The media metadata list must be updated whenever a new media source is synchronized. To facilitate this, the `car-media` module spawns a monitor thread that listens to the `mm-detect` service to be notified when new media sources are connected or synchronized.

The `dialer` module

The `dialer` module allows the user to initiate a phone call by speaking either the number to dial or the name of a known contact. For voice dialing to work, the handsfree phone (HFP) subsystem must be running and a mobile phone device must be paired. If these conditions aren't met, the `dialer` conversation module will still work, but it won't be possible to connect any calls.

Like all other modules, the `dialer` module provides an initialization callback function. After `io-asr` has loaded all modules, it invokes the initialization callback to set up the private data structures managed by the module.

The initialization callback loads the `dialer`-specific configuration tree to make it available to the module. This configuration data is assumed to be in the `module/cardialer` path of the global configuration tree. The `dialer` module has two locale-specific configuration files that are used to customize its behavior:

- `car-control.cfg`—defines text keys that are used to tag terminals in a recognition result. You can add more keys to this section to trigger dialing commands with custom keywords.
- `car-prompts.cfg`—defines the locale-specific prompts that are used by the `dialer` module.

Next, the initialization callback loads the result confidence threshold and opens a connection to the PPS `control` object of the HFP subsystem. The URI of this control object is configurable; the module obtains the location when it loads the configuration data.

When `io-asr` shuts down, it releases all modules that it has dynamically loaded. The `dialer` module provides the *destroy()* callback, which releases resources allocated during the initialization phase and closes any open PPS connections before the module is deleted from the ASR subsystem.

The HFP subsystem

The `dialer` module is responsible for issuing dial commands to the HFP subsystem to initiate outgoing phone calls. Because it relies on the HFP subsytem, the `dialer` module queries the `/pps/services/bluetooth/services` object to ensure the system is available before issuing any commands. It issues dial commands via the PPS handsfree `control` and `status` objects. For more information about these objects, see the *PPS Objects Reference*.

Actions

The `dialer` conversation module can:

- dial a number
- call a named contact
- end an active call
- redial the last number called

In addition to these actions, an `IDLE` action is defined. This describes the system at the beginning of a speech session. Most interactions with the `dialer` will be initiated from this state.

Conversation flow

The `dialer` module has the following conversation states:

- `IDLE`—the default state of the `dialer` module. From this state, the `dialer` module can accept dial, redial, or hangup instructions.
- `DIAL`—the `dialer` module expects a contact name or phone number to call via the HFP subsystem. In this case, the module inspects the result to see if it matches a number to dial. If the result doesn't correspond to a number, the `dialer` module assumes a contact name and searches the contact database for matching information.
- `CONFIRM`—the `dialer` module is ready to dial a number, but first issues a query to the user to confirm whether to dial the number. If the answer is negative, then the `dialer` goes back to the `DIAL` state, prompting the user to provide a number.
- `UNDEFINED_ACTION`—The `dialer` module received an unrecognized command. This state ends the recognition session, setting the result action type (*`asr_result_action_t`* (p. 218)) to `ASR_RECOGNITION_UNKNOWN`.

ASR state transitions

Like other conversation modules, the `dialer` module defines a *step()* callback function to keep track of the current ASR state.

In addition, the `dialer` module's private data structure contains fields that specify the number of digits in the phone number that have been confirmed as correct and

the number of digits that remain. A minimum number of confirmed digits are required before a call can be dialed. The total number of digits in the phone number can be obtained by taking the sum of confirmed digits and new digits.

The *step()* callback function handles the following state transitions:

- ASR_STEP_LOCALE_CHANGED—the configuration of the module may need to change. The *step()* function ensures that the correct locale configuration information is associated with the module's private data. Localized data includes grammar configuration and prompt data.
- ASR_STEP_SESSION_OPENED—places the `dialer` module into the `IDLE` state and sets the base configuration section as active.
- ASR_STEP_RECOGNITION_BEGIN—ensures that the correct configuration section is set as active when a speech-recognition session is started. If the module isn't in the idle state, the confirmation section is set as active; otherwise, the base section is set as active.
- ASR_STEP_RECOGNITION_END—unlocks the phonebook mutex.
- ASR_STEP_SESSION_CLOSED—stops any dialing activity that is currently taking place and removes the module's status as the exclusive recipient of speech commands in the ASR subsystem. To allow the last number to be redialed, this state doesn't clear the previously captured phone number.

The *step()* function ignores all other state transitions.

Result handling

The *on_result()* callback function of the conversation module interface, `asr_conversation_if` (p. 168), handles speech results (utterances) for the digit dialing context. This handler takes one of the following actions:

- initiate a phone call, via a paired Bluetooth handset, to a specified number
- confirm a number to call
- redial the number of any calls previously placed
- terminate a phone call (hang up)

When the utterance has been processed, the handler returns the ASR action (*`asr_result_action_t`* (p. 218)) to perform next. This action indicates whether recognition is complete (the call has been dialed), recognition needs to be restarted (to gather more information and continue the conversation), or recognition should be aborted (terminate the conversation without invoking any functions).

The result-handling callback sets the internal state of the `dialer` module based on the intents of the utterances received and processed by the recognizer. To support incomplete data from an utterance—for example, if a dial request was made without specifying the number, requiring ASR to prompt the user for the number to dial—a

member of the `dialer` module's private data structure keeps track of the internal state between invocations of callbacks.

Grammars

The conversation flow understood by the `dialer` module is defined in the control configuration file by a BNF grammar that expresses the intents of the spoken utterances. The ASR subsystem uses this grammer to extract the intents from the utterance. It adds this information as payload data to the result structure for the NLP component of the conversation pipeline to use. The BNF grammar for the `dialer` module is defined in `car-control.cfg`.

In addition to the NLAL grammar used by the conversation pipeline, the `dialer` module also defines a BNF grammar that can be used to create a context for the third-party recognizer module. The default dialing context is defined in the *`recognizer`*`/en-US/dialer/car-dialer.bnf` directory.

Finally, a default grammar for the contact list is generated. This grammar defines the *contact-name* slot as being a null feature. The context created by this grammar will be replaced when a phonebook is actually connected to the host device. This context simply serves as a placeholder, allowing the `dialer` module to work until the contact list context is generated. This substitution is done at runtime as described in the following section.

Updating the guest context

The `dialer` module defines several helper functions that are used to construct an internal representation of the contact list used to build a guest context for voice dialing. The names of all the contacts are first obtained from the phonebook database and added to an ordered list, removing all duplicate entries.

The contact list associated with the `dialer` module is constructed a little differently than one might expect. Normally, given names are paired with family names. However, in this case, given names and family names are included as separate entries in the contact list. This will allow the constructed context to match on utterances of given name or family name only.

The contact list must be updated whenever there is a change to the phonebook database. To facilitate this, the `dialer` module spawns a monitor thread that listens for changes to the phonebook and updates the contact name list accordingly.

Receive-list database table

The `dialer` conversation module uses a **receive list database table** in the phonebook database to track contacts that are accumulated by the `call-by-name` action. This table (named `asr-matched-contacts`) tracks the contact names and the confidence with which those names correspond to a specified match string. The table can be

easily published to other applications that can access the database (e.g., a GUI application). However, this approach assumes that the database facilities are available and that an appropriate table for storing the data exists. When the module queries the contacts database table, it retrieves the contact ID that is associated with the contact. This allows the module to insert a new row into the table for each match.

Adding a conversation module

To add a new conversation module, create a new instance of the conversation module interface (`asr_conversation_if` (p. 168)), which provides the communication mechanism between your new module and ASR. You must implement the following callback functions, which ASR invokes via the conversation module interface:

- *init()*—initializes the module; invoked by *asr_module_initialize()*.
- *select_result()*—allows modules to accept a recognition result. ASR calls *select_result()* for the current exclusive module if there is one; otherwise, it makes the call for all active registered modules. Each module examines the list of results containing the hypotheses from the current recognition to determine the one it can successfully take action on, and then returns its selection via a result pointer. If there's no exclusive module, the selected result with the highest confidence level is used.
- *on_result()*—handles the selected result. The module takes the appropriate action on the result (for example, invoking an application, dialing a phone number, or reprompting) and returns the next action ASR should take (see *asr_result_action_t* (p. 218) for the list of actions).

  ASR calls *on_result()* for a module only if no other module has a result with a higher confidence level. ASR selects the module that provides the highest rating and sends it the current result for processing. The module can optionally set itself as exclusive by calling the *asrm_set_exclusive()* (p. 92) function so that it will receive all future results in the current speech session (i.e., `io-asr` won't evaluate the ratings of other conversation modules). The module releases its exclusive status when ASR transitions to the `ASR_STEP_SESSION_CLOSED` state.
- *stop()*—for interrupting an action. This function provides a mechanism to interrupt an action and to allow ASR to recover gracefully.

Modules are loaded as DLLs at runtime. You must define a constructor function (designated with the `((constructor))` compiler attribute) that is called when ASR loads your module. This constructor function must call *asrm_connect()* (p. 68) to connect to `io-asr` and obtain your module's handle. The following code fragment illustrates this procedure for the dialer module:

```
typedef struct CarDialer_data_s {
    asr_module_hdl_t*    handle ;
    ...
}CarDialer_data _dialer_data ;

static const asr_conversation_if_t
CarDialer_interface = {
    .name           = "dialer",
    .asr_version    = ASR_VERSION,
    .init           = CarDialer_init,
    .destroy        = CarDialer_destroy,
```

```
    .on_asr_step    = CarDialer_step,
    .select_result  = CarDialer_selectResult,
    .on_result      = CarDialer_onResult,
} ;

__attribute__((constructor))
static void
CarDialer_register( void )
{
    CarDialer_data* self = &_dialer_data ;
    asr_module_hdl_t* module_handle ;

    memset( self, 0, sizeof( *self ) ) ;
    module_handle = asrm_connect( &CarDialer_interface, sizeof( CarDialer_interface ), self ) ;
    if ( self && module_handle ) {
            self->handle = module_handle ;
}
}
```

In this example, the *self* data pointer is associated with the module. The `io-asr` service passes this pointer to all the module's callbacks as `void*` arguments. The pointer is then cast to the proper structure type to provide access to the module-specific data.

When ASR unloads the conversation module, it calls the module's *destroy()* callback function. This function should clean up any resources that were allocated in the *init()* function.

Specifying NLAL grammars

Depending on the functionality the new module is supporting, you may also need to create a new configuration file for the conversation. The configuration file defines the grammar that is used to interpret the text and to allow the module to take the appropriate action.

In the configuration file for each conversation module (${***QNX_TARGET***}/opt/asr/conversation/*locale/module*/car-control.cfg), define a section with the following syntax:

```
section-name = NLAL {
 }
```

This specifies `section-name` as the top-level grammar rule for the module. Each line of this configuration section describes an alternate grammar that the NLAL uses to match an utterance. You can define any number of these sections, and each section can contain any number of entries. In the code for the new conversation module, you call the *asrm_set_active_sections()* (p. 91) function to set the grammar that the module uses. The top-level rules can be composed of subrules. You specify a subrule with angled brackets. For example:

```
time-query = {
        'what is the current time'
}

keys = NLAL {
```

```
        <time-query>
}
```

In this configuration, the `<time-query>` rule is expanded to the contents of the `time-query` subrule, which is composed of string literals. Therefore, the only utterance that will match this rule is the sentence "what is the current time". Note that you don't use the `NLAL` attribute for subrules; it's only required for the top-level rules.

You can also add actions to the rule. For example, the `keys` section could be extended as follows:

```
keys = NLAL {
        <time-query:id(rule, time)>
}
```

This has the effect that whenever an utterance matches the `<time-query>` rule, an intent with the field name `rule` and the value `time` will be added to the result structure (*asr_result* (p. 205)):

```
intents = {
        rule = time
}
```

If the value is omitted from the `id` command, the matching text is set as the field value. For example:

```
search-query = {
        'search for '<...:id(search-term)>
}

keys = NLAL {
        <search-query:id(rule,search)>
}
```

This rule matches utterances of the form "search for ...". The special rule `<...>` is a catch-all that matches anything. If the NLAL processes the utterance "Search for Starbucks" using this rule, the following intent fields will be added to the result structure:

```
intents = {
        rule = search
        search-term = Starbucks
}
```

It's also possible to optionally match items in the grammar by enclosing them in square brackets. The rule:

```
search-query = {
        search ['internet '] 'for ' <...:id(search-term)> }
```

matches the utterances:

- "Search for Starbucks"
- "Search internet for Starbucks"

Note that spaces must be quoted and must be described explicitly in the rule. The following rule is incorrect:

```
search-query = {
        search [internet] for <...:id(search-term)> }
```

This rule won't match either of the previously described utterances. However, it will match the following:

- “searchinternetforStarbucks”
- “searchforStarbucks”

Items enclosed in parentheses and separated by pipe characters specify alternative matches. For example, the rule:

```
search-query = {
  'search ' ( internet | media ) ' for ' <...:id(search-term)> }
```

matches the utterances:

- “search internet for Weezer”
- “search media for Weezer”
- “search for Weezer”

This has the same effect as defining a subrule with a different match grammar on each line:

```
search-type = {
        internet
        media
}

search-query = {
        'search ' <search-time> ' for ' <...:id(search-term)> }
```

You can add the special `help-url` item to the top-level NLAL section rule to define a prompt string for ASR to use to respond to user help requests. If the user utters one of the help commands (defined in the common configuration file, ${***QNX_TARGET***}`/opt/asr/conversation/`*`locale`*`/common/car-control.cfg`) while this section is active, ASR passes the value of the `help-url` to the prompt module:

```
keys = NLAL {
   <search-query:id(rule, search)>
   help-url = tts://"Describe the commands supported by this grammar"
}
```

In this example, the if the user requests help while the `keys` section is active, ASR passes the the URL `tts://"Describe the commands supported by this grammar"` to the prompt module.

Chapter 3
API Reference

The following table summarizes the header files that provide the ASR API:

Header file	Description
`asr.h`	Provides an interface to the host system's logging facilities.
`asra.h`	Provides functions and data types for capturing and playing back audio.
`asrm.h`	Provides functions and data types for module management.
`asrp.h`	Provides functions and data types for rendering prompts.
`asrv.h`	Provides functions to interact with vendor-supplied modules.
`cfg.h`	Contains data types and functions for building and searching a configuration tree.
`mod_types.h`	Provides data type definitions and functions for the audio, recognition, and conversation modules.
`protos.h`	Provides functions for logging.
`slot-factory.h`	Provides functions and data types for interacting with the `SlotFactory` object, which is used to manage recognition results.
`terminals.h`	For internal use.
`types.h`	Provides data types for the control flow of speech recognition. These data types include result classifications, state enumerations, and error codes.

asr.h

Functions and data types for automatic speech recognition.

The `asr.h` header file provides functions and data types for interacting with io-asr.

Definitions in *asr.h*

Preprocessor macro definitions for the asr.h header file in the libasr library.

Definitions:

```
#define ASR_RESPONSE_PENDING 0x8000
```

A flag to indicate that a recognition response is pending.

Library:

`libasr`

Data types in *asr.h*

asr_context_hdl

The context handle.

Synopsis:

```
struct asr_context_hdl {
    int dummy ;
};
```

Data:

int dummy

A dummy field that can be redeclared depending on vendor requirements.

Library:

`libasr`

Description:

This opaque data type represents the context handle, which is an opaque definition of a local recognizer context. The context is used to add new words to a recognizer, increasing the chance that it will be able to find correct matches for more utterances. The required structure of this data type may vary by vendor.

asr_global_data_t

Information about the recognizer module.

Synopsis:

```
#include "asr/asr.h"

typedef struct asr_global_data asr_global_data_t;
```

Library:

`libasr`

Description:

This opaque data type carries global information about the recognizer module. This information may vary by vendor.

asr_instance_data_t

Identifying information about an instance of the recognition service.

Synopsis:

```
#include "asr/asr.h"

typedef struct asr_instance_data asr_instance_data_t;
```

Library:

`libasr`

Description:

This opaque data type carries identifying information about an instance of the speech recognition service. This information may vary by vendor.

Functions in *asr.h*

asr_cancel()

Cancel a recognition request.

Synopsis:

```
#include "asr/asr.h"

int asr_cancel()
```

Arguments:

Library:

`libasr`

Description:

The *asr_cancel*() function invokes the *stop*() callback function defined in the recognizer interface, asr_recognizer_if, and sets the hold count to 0. The recognizer stops audio acquisition and stops processing results.

Returns:

0 Success.

-1 An error occurred; `errno` is set.

asr_close_global()

Close a connection to the recognizer module.

Synopsis:

```
#include "asr/asr.h"

int asr_close_global(void *asr_hdl, asr_global_data_t *data)
```

Arguments:

asr_hdl

The handle to close.

data

Global data to be passed to or received from `io-asr`.

Library:

`libasr`

Description:

The *asr_close_global*() closes a connection to the recognizer module. The operations performed by the *asr_close_global*() function and the contents of the `asr_global_data_t` structure may vary by vendor.

Returns:

0 Success.

-1 An error occurred.

asr_close_instance()

Close an instance of the recognizer module.

Synopsis:

```
#include "asr/asr.h"

int asr_close_instance(void *asr_hdl, asr_instance_data_t *data)
```

Arguments:

asr_hdl

The recognizer handle.

data

The instance data to be freed.

Library:

`libasr`

Description:

The *asr_close_instance*() function closes the specified instance of the recognizer and frees the memory consumed by *data*. The operations performed by the *asr_close_instance*() function and the contents of the `asr_instance_data_t` structure may vary by vendor.

Returns:

0 on success; an error code on error.

asr_context_add_entries()

Add entries to the specified context.

Synopsis:

```
#include "asr/asr.h"

int asr_context_add_entries(asr_context_hdl_t *chdl, cfg_item_t *cfg, const
char *slot_identifier, asr_slot_entry_t *slot_entry, int num_slot_entries)
```

Arguments:

chdl

A pointer to the context handle.

cfg

A pointer to the configuration associated with the context.

slot_identifier

A pointer to the slot identifier (the position of the new entry).

slot_entry

An array of slot entries.

num_slot_entries

The size of the array of slot entries.

Library:

`libasr`

Description:

The *asr_context_add_entries*() function invokes the *context_add_entries*() callback function defined in the recognizer interface, asr_recognizer_if.

Returns:

0 Success.

<0 An error occurred.

asr_context_create()

Create a conversation context.

Synopsis:

```
#include "asr/asr.h"

asr_context_hdl_t* asr_context_create(cfg_item_t *cfg)
```

Arguments:

cfg

A pointer to the configuration item associated with the context.

Library:

`libasr`

Description:

The *asr_context_create*() function invokes the *context_create*() callback function defined in the recognizer interface, asr_recognizer_if.

Returns:

A pointer to the new context handle on success; NULL on error.

asr_context_destroy()

Destroy a conversation context.

Synopsis:

```
#include "asr/asr.h"

int asr_context_destroy(asr_context_hdl_t *chdl)
```

Arguments:

chdl

A pointer to the context handle.

Library:

`libasr`

Description:

The *asr_context_create*() function invokes the *context_create*() callback function defined in the recognizer interface, asr_recognizer_if.

Returns:

0 Success.

-1 An error occurred; `errno` is set.

asr_context_save()

Save a context.

Synopsis:

```
#include "asr/asr.h"

int asr_context_save(asr_context_hdl_t *chdl, cfg_item_t *cfg)
```

Arguments:

chdl

The context handle.

cfg

The configuration structure for the recognizer.

Library:

`libasr`

Description:

The *asr_context_save*() function invokes the *context_save*() callback function defined in the recognizer interface, asr_recognizer_if.

Returns:

0 Success.

<0 An error occurred.

asr_get_hold_count()

Return the number of holds on the recognizer.

Synopsis:

```
#include "asr/asr.h"

int asr_get_hold_count()
```

Arguments:

Library:

`libasr`

Description:

The *asr_get_hold_count*() function returns the number of holds on the recognizer.

Returns:

The number of holds.

asr_get_restart()

Return the recongizer restart setting.

Synopsis:

```
#include "asr/asr.h"

int asr_get_restart()
```

Arguments:

Library:

`libasr`

Description:

The *asr_get_restart*() function returns the recognizer restart setting.

Returns:

The recognizer restart setting.

asr_get_utterance()

Capture an utterance.

Synopsis:

```
#include "asr/asr.h"

int asr_get_utterance(asr_audio_info_t *audio_info)
```

Arguments:

audio_info

Audio capture information

Library:

`libasr`

Description:

The *asr_get_utterance*() function invokes the *get_utterance*() callback function defined in the recognizer interface, asr_recognizer_if. If no *get_utterance*() function is defined in the recognizer interface, the *get_utterance*() callback function defined in the audio interface, asra_module_interface, is invoked instead.

Returns:

0 Success.

<0 An error occurred.

asr_hold()

Place a hold on the recognizer.

Synopsis:

```
#include "asr/asr.h"

int asr_hold()
```

Arguments:

Library:

`libasr`

Description:

The *asr_hold*() function invokes the *stop*() callback function defined in the recognizer interface, asr_recognizer_if, and increments the recognizer hold count. The recognizer stops acquiring audio and processing results for the current request.

Returns:

The number of holds.

asr_open_global()

Open a connection to the recognizer module.

Synopsis:

```
#include "asr/asr.h"

int asr_open_global(void *asr_hdl, cfg_item_t *config_base, asr_global_data_t
 **data)
```

Arguments:

asr_hdl

The handle to the recognizer module.

config_base

Configuration data for the recognizer.

data

Identifying information about the recognizer.

Library:

`libasr`

Description:

The *asr_open_global*() function initializes the recognizer and returns identifying information about the recognizer via the *data* parameter. The operations performed by the *asr_open_global*() function and the contents of the `asr_global_data_t` structure may vary by vendor.

Returns:

0 Success.

-1 An error occurred.

asr_open_instance()

Open an instance of the recognizer module.

Synopsis:

```
#include "asr/asr.h"

int asr_open_instance(void *asr_hdl, asr_instance_data_t **data)
```

Arguments:

asr_hdl

The handle to the recognizer service.

data

Identifying information about the instance.

Library:

`libasr`

Description:

The *asr_open_instance*() function opens a new instance of the recognizer and returns identifying information about it via the *data* parameter. The operations performed by the *asr_open_instance*() function and the contents of the `asr_instance_data_t` structure may vary by vendor.

Returns:

0 on success; an error code on error.

asr_post_step()

Process a state change.

Synopsis:

```
#include "asr/asr.h"

void asr_post_step(asr_step_t step)
```

Arguments:

step

The step to handle.

Library:

`libasr`

Description:

The *asr_post_step*() function invokes the *step*() callback function defined in the recognizer interface, asr_recognizer_if.

Returns:

Nothing.

asr_recognition_initialize()

Initialize the recognizer module.

Synopsis:

```
#include "asr/asr.h"

int asr_recognition_initialize()
```

Arguments:

Library:

`libasr`

Description:

The *asr_recognition_initialize*() function invokes the *init*() callback function defined in the recognizer interface, asr_recognizer_if, for each active recognizer module.

Returns:

0 Success.

<0 An error occurred.

asr_release()

Release a hold on the recognizer.

Synopsis:

```
#include "asr/asr.h"

int asr_release()
```

Arguments:

Library:

`libasr`

Description:

The *asr_release*() function reduces the hold count on the recognizer by one. If no holds remain it starts the recognizer.

Returns:

The number of holds remaining.

asr_reload_localization()

Reload localization information.

Synopsis:

```
#include "asr/asr.h"

int asr_reload_localization(void)
```

Arguments:

Library:

`libasr`

Description:

The *asr_reload_localization*() function finds modules that require localized assets and reloads the definitions for those assets from the configuration structure.

Returns:

0 Success.

<0 An error occurred.

asr_result_map_status()

Map a vendor-specific recognition result status to a generic ASR result status.

Synopsis:

```
#include "asr/asr.h"

int asr_result_map_status(void *vendor_AsrRes)
```

Arguments:

vendor_AsrRes

The vendor-specific result status.

Library:

`libasr`

Description:

The *asr_result_map_status*() maps a results status and confidence level to a member of the `result_status` enumeration. The exact mapping is vendor dependent.

Returns:

A corresponding generic ASR result status from the `result_status` enumeration. Note that a return code of ASR_RESULT_OK means that an exact mapping wasn't successful.

asr_set_recognizer()

Set a recognizer as current.

Synopsis:

```
#include "asr/asr.h"

int asr_set_recognizer(const char *recognizer)
```

Arguments:

recognizer

The name of the recognizer to set as current.

Library:

`libasr`

Description:

The *asr_set_recognizer*() function sets the specified recognizer as the current one for handling recognition requests.

Returns:

0 Success.

<0 An error occurred.

asr_set_restart()

Set the recongizer restart setting.

Synopsis:

```
#include "asr/asr.h"

void asr_set_restart(int restart)
```

Arguments:

restart

The recognizer restart setting.

Library:

`libasr`

Description:

The *asr_get_restart*() function sets the recognizer restart setting.

Returns:

Nothing.

asr_set_utterance()

Copy an utterance to the specified buffer.

Synopsis:

```
#include "asr/asr.h"

int asr_set_utterance(asr_audio_info_t *audio_info, uint32_t ms_offset)
```

Arguments:

audio_info

Indicates the structure in which to store the utterance.

ms_offset

The offset (in milliseconds) of the utterance.

Library:

`libasr`

Description:

The *asr_set_utterance*() function invokes the *set_utterance*() callback function defined in the recognizer interface, asr_recognizer_if. If no *set_utterance*() function is defined in the recognizer interface, the *set_utterance*() callback function defined in the audio interface, asra_module_interface, is invoked instead.

Returns:

0 Success.

EBUSY Capture has not completed.

EINVAL The audio properties don't match.

ERANGE Buffer overrun.

asr_slog()

Capture logging information for the recognizer.

Synopsis:

```
#include "asr/asr.h"

int asr_slog(asr_recognizer_hdl_t *mod, int severity, const char *fmt,...)
__attribute__((format(printf
```

Arguments:

mod

The recognizer handle.

severity

The severity of the condition that triggered the message. For more information on severity levels, see *slogf()* in the *QNX C Library Reference*. Valid values include:

- `_SLOG_INFO`
- `_SLOG_WARN`
- `_SLOG_ERROR`
- `_SLOG_CRITICAL`

fmt

The format string to print to the log buffer. This may include tokens that will be replaced by values of variable arguments appended to the end of the call. The max length of an expanded log message is 1024 characters (this includes all format substitutions and the null terminator).

Library:

`libasr`

Description:

The *asr_slog*() function sends debugging information to the appropriate log. Log messages will be written to the log buffer only if their severity is greater than or equal to the specified severity.

Returns:

0 Success.

<0 An error occurred.

asr_start()

Start a recognition request.

Synopsis:

```
#include "asr/asr.h"

int asr_start()
```

Arguments:

Library:

`libasr`

Description:

The *asr_start*() function starts a recognition request by invoking the *start*() callback function defined in the recognizer interface, asr_recognizer_if. The recognizer should collect and process the audio sample, and then provide status and results via the API defined in the ASR vendor interface, `asrv.h`. This call must be asynchronous and the recognition operation started must be interuptable via a call to *asr_stop()*.

Returns:

0 Success.

<0 An error occurred.

asr_stop()

Stop an in-process recognition request.

Synopsis:

```
#include "asr/asr.h"

int asr_stop()
```

Arguments:

Library:

`libasr`

Description:

The *asr_stop*() function invokes the *stop*() callback function defined in the recognizer interface, asr_recognizer_if, and sets the hold count to 0. The recognizer stops audio acquisition and stops processing results. This call blocks until the recognizer returns, confirming that the recognition request has terminated. If there's no recognition request running, *asr_stop*() returns immediately with a successful result.

Returns:

0 Success.

-1 An error occurred; `errno` is set.

asra.h

Data types and functions for interacting with the audio module.

The `asra.h` header file provides functions and data types for capturing audio from the microphone or reading audio data from a file.

Data types in *asra.h*

asra_module_hdl_t

This audio module handle.

Synopsis:

```
#include "asr/asra.h"

typedef struct asra_module_hdl asra_module_hdl_t;
```

Library:

`libasr`

Description:

This opaque type represents the audio module handle and is used by `io-asr` to manage data it passes to and from the audio module.

Functions in *asra.h*

asr_audio_initialize()

Initialize the audio module.

Synopsis:

```
#include "asr/asra.h"

int asr_audio_initialize(void)
```

Arguments:

Library:

`libasr`

Description:

The *asr_audio_initialize*() function initializes registered audio modules by invoking their *init*() callback functions (see asra_module_interface_t).

Returns:

0 Success.

-1 An error occurred.

asra_acquire_buffer()

Request an audio buffer.

Synopsis:

```
#include "asr/asra.h"

int asra_acquire_buffer(asr_audio_info_t *info, int wait)
```

Arguments:

info

The structure to store the audio sample.

wait

An optional flag to indicate whether the module should wait for a successful audio sample.

Library:

`libasr`

Description:

The *asra_acquire_buffer*() function requests a buffer.

Returns:

0 Capturing has finished. The buffer is available.

>0 Capturing is ongoing.

<0 An error ocurred.

asra_close()

Close the audio module.

Synopsis:

```
#include "asr/asra.h"

int asra_close()
```

Arguments:

Library:

`libasr`

Description:

The *asra_close*() function closes the current audio module.

Returns:

0 Success.

<0 An error occurred.

asra_connect()

Connect to the audio module.

Synopsis:

```
#include "asr/asra.h"

asra_module_hdl_t* asra_connect(const asra_module_interface_t *aif, unsigned
len)
```

Arguments:

aif

The audio module interface.

len

The size of the audio module interface.

Library:

`libasr`

Description:

The *asra_connect*() function connects to the specified audio module by adding the module to `io-asr`'s list of current modules.

Returns:

The audio module handle on success; NULL on error, with error details written to the log.

asra_disconnect()

Disconnect the prompt module.

Synopsis:

```
#include "asr/asra.h"

void asra_disconnect(asra_module_hdl_t *hdl)
```

Arguments:

hdl

The prompt module handle.

Library:

`libasr`

Description:

The *asra_disconnect*() function disconnects the specified audio module from `io-asr` and frees the associated memory.

Returns:

Nothing.

asra_get_utterance()

Capture an utterance.

Synopsis:

```
#include "asr/asra.h"

int asra_get_utterance(asr_audio_info_t *info)
```

Arguments:

info

The structure in which to store the utterance and set the properties.

Library:

`libasr`

Description:

The *asra_get_utterance*() function stores an audio sample in the buffer referenced by the *info* parameter. It also sets the associated properties of the utterance: buffer size, sample size, sample rate, and number of channels. It waits until the audio capture has completed before copying the sample and returning.

Returns:

0 Success.

-1 An error occurred.

asra_open()

Open the audio module.

Synopsis:

```
#include "asr/asra.h"

int asra_open()
```

Arguments:

Library:

`libasr`

Description:

The *asra_open*() function opens the current audio module.

Returns:

>=0 Success.

<0 An error occurred.

asra_relinquish_buffer()

Relinquish an audio buffer.

Synopsis:

```
#include "asr/asra.h"

int asra_relinquish_buffer(asr_audio_info_t *info)
```

Arguments:

info

The structure that contains the buffer.

Library:

`libasr`

Description:

The *asra_relinquish_buffer*() function resets the buffer in the *info* structure so that it can be used again.

Returns:

0 Success.

-1 An error occurred.

asra_save_wavefile()

Save the captured audio sample as a WAV file.

Synopsis:

```
#include "asr/asra.h"

int asra_save_wavefile(const char *fname)
```

Arguments:

fname

The name to use for the WAV file.

Library:

`libasr`

Description:

The *asra_save_wavefile()* copies the captured audio sample as a WAV file with the specified filename.

Returns:

0 Success.

-1 The file couldn't be opened for writing.

asra_set_params()

Set the audio parameters for the module.

Synopsis:

```
#include "asr/asra.h"

int asra_set_params(int sample_rate, int volume, int frag_size)
```

Arguments:

sample_rate

The audio sample rate.

volume

The audio volume.

frag_size

The audio fragment size.

Library:

`libasr`

Description:

The *asra_set_params*() function sets the global audio parameters.

Returns:

0 Success.

-1 An error occurred.

asra_set_source()

Set the audio source.

Synopsis:

```
#include "asr/asra.h"

int asra_set_source(const char *url)
```

Arguments:

url

The audio source URL.

Library:

`libasr`

Description:

The *asra_set_source*() sets the specified URL as the audio source and sets the module best suited to that URL (the module that rates itself highest) as current.

Returns:

0 Success.

-1 An error occurred.

asra_set_utterance()

Copy an utterance to the specified buffer.

Synopsis:

```
#include "asr/asra.h"

int asra_set_utterance(asr_audio_info_t *info, int offset_ms)
```

Arguments:

info

Indicates the structure in which to store the utterance.

offset_ms

The offset (in milliseconds) of the utterance.

Library:

`libasr`

Description:

The *asra_set_utterance*() function copies the last captured audio sample to the buffer referenced by the *info* parameter, at the offset specified by the *offset_ms* parameter. If the requested offset results in a buffer overrun, an error is returned. If the audio capture hasn't completed, an error is returned.

Returns:

0 Success.

EBUSY Capture has not completed.

ENOMEM Buffer overrun or other memory error.

asra_start()

Start the audio module.

Synopsis:

```
#include "asr/asra.h"

int asra_start()
```

Arguments:

Library:

`libasr`

Description:

The *asra_start*() function causes the module to begin to perform its particular service, for example capturing audio or playing back from a file.

Returns:

0 Success.

-1 An error occurred.

asra_stop()

Stop the audio capture.

Synopsis:

```
#include "asr/asra.h"

int asra_stop()
```

Arguments:

Library:

`libasr`

Description:

The *asra_stop*() function forces the audio capturing to stop.

Returns:

0 Success.

-1 An error occurred.

asrm.h

Functions and data types for module management.

The `asrm.h` header file provides functions and data types for module management. It also includes support functions used to implement conversation flows (e.g., verifying whether a result corresponds to a help request or a cancel request).

Enumerations in *asrm.h*

asrm_phrase_search_mode_t

Phrase search mode enumeration.

Synopsis:

```
#include "asr/asrm.h"


 enum {
      PHRASE_EXACT == 0x01
      PHRASE_FUZZY == 0x02
} asrm_phrase_search_mode_t;
```

Data:

PHRASE_EXACT

Exact match searching.

PHRASE_FUZZY

Fuzzy match searching.

Library:

`libasr`

Description:

The *asrm_phrase_search_mode_t* enumeration lists the modes of phrase searching.

Functions in *asrm.h*

asr_module_initialize()

Initialize all registered modules.

Synopsis:

```
#include "asr/asrm.h"

int asr_module_initialize(void)
```

Arguments:

None.

Library:

`libasr`

Description:

The *asr_module_initialize*() function initializes all registered modules by invoking their respective *init*() callback functions.

Returns:

0 Success.

-1 A module couldn't be initialized. The details are written to the log.

asr_strmatch()

Calculate the confidence that two strings match.

Synopsis:

```
#include "asr/asrm.h"

int asr_strmatch(const char *str1, const char *str2)
```

Arguments:

str1

The first string to compare.

str2

The second string to compare.

Library:

`libasr`

Description:

The *asr_strmatch*() function calculates a confidence score that can be used to evaluate the confidence with which *str1* matches *str2*. Higher confidence scores indicate a higher confidence that the two strings match. This algorithm is based on the Damerau-Levenshtein *edit distance* algorithm.

Returns:

An integer confidence score (0 - 1000) that indicates the confidence with which *str1* and *str2* match.

asrm_activate_module()

Set a module as active.

Synopsis:

```
#include "asr/asrm.h"

void asrm_activate_module(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_activate_module*() function sets the specified module as active.

Returns:

Nothing.

asrm_append_intent()

Append an intent to the specified result.

Synopsis:

```
#include "asr/asrm.h"

void asrm_append_intent(asr_result_t *result, char *key, char *value)
```

Arguments:

result

The result to add the intent to.

key

The key of the intent.

value

The value of the intent.

Library:

`libasr`

Description:

The *asrm_append_intent*() function appends the intent specified by *key* and *value* to the intents array of the specified *result*. Note that sufficient memory must be available to add items to the intent structure. No warning or error is generated if there is insufficient memory.

Returns:

Nothing.

asrm_append_result()

Append a result to a list of results.

Synopsis:

```
#include "asr/asrm.h"

asr_result_t* asrm_append_result(asr_result_t *results, asr_result_t
*new_result)
```

Arguments:

results

The results list.

new_result

The result to append.

Library:

`libasr`

Description:

The *asrm_append_result*() function appends the result specified by *new_result* to the specified results list, *results*.

Returns:

A pointer to the updated results list.

asrm_connect()

Connect to the module.

Synopsis:

```
#include "asr/asrm.h"

asr_module_hdl_t* asrm_connect(const asr_conversation_if_t *cif, unsigned len,
 void *module_private)
```

Arguments:

cif

The conversation module interface.

len

The size of the conversation module interface.

module_private

Module-specific data that can be attached to the module. The `io-asr` service passes this data to to the module's callback functions to support module-specific actions.

Library:

`libasr`

Description:

The *asrm_connect*() function connects to the specified module by adding the module to `io-asr`'s list of current modules.

Returns:

The audio module handle on success; NULL on error, with error details written to the log.

asrm_context_add_entries()

Add context entries.

Synopsis:

```
#include "asr/asrm.h"

int asrm_context_add_entries(asr_context_hdl_t *chdl, const char
*slot_identifier_section, const char *slot_name, asr_slot_entry_t *slot_entry,
 int num_slot_entries)
```

Arguments:

chdl

A pointer to the handle of the context to add entries to.

slot_identifier_section

The configuration node for the section required (e.g., "phone").

slot_name

The name of the slot to add (e.g., "voice dialing").

slot_entry

An array of slot entries.

num_slot_entries

The size of the array of slot entries.

Library:

`libasr`

Description:

The *asrm_context_add_entries*() function adds entries to the specified context by first finding the configuration node identified by the *slot_identifier_section*, then invoking the current recognition module's *context_add_entries*() callback function on that configuration node. The exact implementation of the *context_add_entries*() callback function depends on the ASR vendor.

Returns:

0 Success.

<0 An error occurred.

asrm_context_create()

Create a context.

Synopsis:

```
#include "asr/asrm.h"

asr_context_hdl_t* asrm_context_create(const char *section_identifier)
```

Arguments:

section_identifier

The configuration section to use to create the context.

Library:

`libasr`

Description:

The *asrm_context_create*() function creates a new context from the specified section by invoking the current recognition module's *context_create*() callback. The exact implementation of the callback depends on the ASR vendor.

Returns:

The new context handle.

asrm_context_destroy()

Save a context.

Synopsis:

```
#include "asr/asrm.h"

int asrm_context_destroy(asr_context_hdl_t *chdl)
```

Arguments:

chdl

A pointer to the context handle.

Library:

`libasr`

Description:

The *asrm_context_destroy*() function destroys the specified context by invoking the current recognition module's *context_destroy*() callback function. The exact implementation of the callback depends on the ASR vendor.

Returns:

0 Success.

<0 An error occurred.

asrm_context_save()

Save a context.

Synopsis:

```
#include "asr/asrm.h"

int asrm_context_save(asr_context_hdl_t *chdl, const char *section_identifier)
```

Arguments:

chdl

A pointer to the context handle.

section_identifier

The configuration section to save.

Library:

`libasr`

Description:

The *asrm_context_save*() function saves the specified context by first finding the configuration section identified by the *section_identifier*, then invoking the current recognition module's *context_save*() callback function on that section. The exact implementation of the callback depends on the ASR vendor.

Returns:

0 Success.

<0 An error occurred.

asrm_create_dictation_result()

Create a dictation result.

Synopsis:

```
#include "asr/asrm.h"

asr_result_t* asrm_create_dictation_result(const char *grammar, const char
*rule, int conf, const char *inbuffer)
```

Arguments:

grammar

The grammar of the new result.

rule

The rule of the new result.

conf

The confidence level of the new result.

inbuffer

The recognized speech to copy to the new result.

Library:

`libasr`

Description:

The *asrm_create_dictation_result*() creates a new dictation result based on the specified parameters. The new result has the recognition type set to ASR_RECOGNITION_DICTATION, the result type set to ASR_RESULT_FINAL, and the result status set to ASR_RESULT_OK. The grammar, rule, confidence level, and recognized speech are set to the corresponding specified parameters.

Returns:

The new result; NULL on error.

asrm_deactivate_module()

Set a module as inactive.

Synopsis:

```
#include "asr/asrm.h"

void asrm_deactivate_module(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_deactivate_module*() function sets the specified module as inactive.

Returns:

Nothing.

asrm_delete_terminals()

Delete result terminals.

Synopsis:

```
#include "asr/asrm.h"

int asrm_delete_terminals(asr_result_t *result, unsigned int start_index,
unsigned int end_index)
```

Arguments:

result

A pointer to the result structure to delete terminals from.

start_index

The index from which to start the deletion.

end_index

The index of the last terminal to delete.

Library:

`libasr`

Description:

The *asrm_delete_terminals*() deletes the result terminals from *start_index* to *end_index*, inclusive.

Returns:

0 Success.

-1 An error occurred.

asrm_find_module()

Find the specified module.

Synopsis:

```
#include "asr/asrm.h"

asr_module_hdl_t* asrm_find_module(const char *module_name)
```

Arguments:

module_name

The name of the module to find.

Library:

`libasr`

Description:

The *asrm_find_module*() function returns the module handle of the named module.

Returns:

A pointer to the module handle; NULL if the module can't be found.

asrm_find_phrase()

Find a configuration item containing a result string.

Synopsis:

```
#include "asr/asrm.h"

cfg_item_t* asrm_find_phrase(cfg_item_t *base, const char *result, int
start_terminal, asrm_phrase_search_mode_t mode, int *beg_terminal, int
*end_terminal, int *conf)
```

Arguments:

base

The configuration structure to search.

result

The result string to search for.

start_terminal

Not used.

mode

The search mode (from the `asrm_phrase_search_mode_t` enumeration).

beg_terminal

On success, *beg_terminal* is set to the index of the first terminal in the match.

end_terminal

On success, *end_terminal* is set to the index of the last terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

Library:

`libasr`

Description:

The *asrm_find_phrase*() function searches a configuration structure for a specified result string and returns a pointer to the matching configuration item.

Returns:

A pointer to the configuration item that was the closest match; NULL on failure.

asrm_find_phrase_id()

Return the ID of a configuration item containing a result string.

Synopsis:

```
#include "asr/asrm.h"

int asrm_find_phrase_id(cfg_item_t *base, const char *result_string, int
start_terminal, int *terminal_end, int *conf, int def_id)
```

Arguments:

base

The configuration structure to search.

result_string

The result string to search for.

start_terminal

Not used.

terminal_end

On success, *terminal_end* is set to the index of the last terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

def_id

A default ID to return if the configuration item isn't found.

Library:

`libasr`

Description:

The *asrm_find_phrase_id()* function searches a configuration for a specified result string and returns the ID of the matching configuration item. The configuration item must contain an exact match.

Returns:

A pointer to the configuration item that was the closest match; NULL on failure.

asrm_find_result_phrase()

Find a configuration item containing a speech result.

Synopsis:

```
#include "asr/asrm.h"

cfg_item_t* asrm_find_result_phrase(cfg_item_t *base, asr_result_t *result,
int start_terminal, int *terminal_beg, int *terminal_end, int *conf)
```

Arguments:

base

The configuration structure to search.

result

The speech result containing the string to search for.

start_terminal

Not used.

terminal_beg

On success, *terminal_beg* is set to the index of the first terminal in the match.

terminal_end

On success, *terminal_end* is set to the index of the last terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

Library:

`libasr`

Description:

The *asrm_find_result_phrase*() function searches the specified configuration for the string in the specified speech result and returns a pointer to the matching configuration item. The configuration item must contain an exact match.

Returns:

A pointer to the matching configuration item; NULL on failure.

asrm_find_result_phrase_id()

Return the ID of a configuration item containing a speech result.

Synopsis:

```
#include "asr/asrm.h"

int asrm_find_result_phrase_id(cfg_item_t *base, asr_result_t *result, int
start_terminal, int *terminal_beg, int *terminal_end, int *conf, int def_id)
```

Arguments:

base

The configuration structure to search.

result

The speech result containing the string to search for.

start_terminal

Not used.

terminal_beg

On success, *beg_terminal* is set to the index of the first terminal in the match.

terminal_end

On success, *beg_terminal* is set to the index of the last terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

def_id

A default ID to return if the phrase isn't found.

Library:

`libasr`

Description:

The *asrm_find_result_phrase_id*() function searches the specified configuration for the string in the specified speech result and returns the ID of the matching configuration item. The configuration item must contain an exact match.

Returns:

The configuration ID of the matching configuration item; the default ID if a match isn't found.

asrm_free_result()

Free the memory associated with a recognition result.

Synopsis:

```
#include "asr/asrm.h"

void asrm_free_result(asr_result_t *result)
```

Arguments:

result

A pointer to the recognition result to free.

Library:

`libasr`

Description:

The *asrm_free_result*() function frees all the memory associated with and referenced by the specified *result*.

Returns:

Nothing.

asrm_get_config()

Get the current configuration.

Synopsis:

```
#include "asr/asrm.h"

cfg_item_t* asrm_get_config()
```

Arguments:

Library:

`libasr`

Description:

The *asrm_get_config*() function retrieves the current configuration tree.

Returns:

A pointer to the root of the configuration tree.

asrm_get_exclusive()

Get the exclusive module.

Synopsis:

```
#include "asr/asrm.h"

asr_module_hdl_t* asrm_get_exclusive(void)
```

Arguments:

Library:

`libasr`

Description:

The *asrm_get_exclusive*() function returns a handle to the exclusive module, if there is one.

Returns:

The exclusive module's handle; NULL if there is no exclusive module.

asrm_get_holdcount()

Return the number of holds on the recognizer.

Synopsis:

```
#include "asr/asrm.h"

int asrm_get_holdcount(void)
```

Arguments:

Library:

`libasr`

Description:

The *asrm_get_holdcount*() function returns the number of holds on the recognizer.

Returns:

The number of holds.

asrm_get_intent_field()

Get the specified intent.

Synopsis:

```
#include "asr/asrm.h"

const char* asrm_get_intent_field(asr_result_t *result, const char *field,
asr_result_tag_t **tag, int *iterator)
```

Arguments:

result

The result structure to search.

field

The field to search for.

tag

The tag entry for the successfully located intent.

iterator

The index to start searching from. On return, the index of the successfully located intent.

Library:

`libasr`

Description:

The *asrm_get_intent_field*() function returns a reference to the specified intent, *field*, within one of the specified result's intent entries. If *tag* is provided, it's set to point to the intent's tag entry, which contains confidence levels, an ID, and possibly millisecond start and end values. If *iterator* is provided, its value is used as a starting point for scanning for the specified intent. If an intent is found whose key matches *field*, its index is stored in *iterator*.

If *field* is NULL, the entry at the index indicated by *iterator* is returned. If *tag* is NULL, no tag information is returned. If iterator is NULL, the index search begins at 0.

For example, to get the "hour" field:

```
value = asrm_get_intent_field (result, "hour", NULL, NULL);
```

To extract all "hour" fields:

```
for (i = 0; (value = asrm_get_intent_field(result, "hour", NULL, &i)); ){
printf ("Found hour intent, value = %s\n", value);
}
-- extract all fields
for (i = 0; (value = asrm_get_intent_field(result, NULL, NULL, &i)); ){
printf ("Found hour intent, value = %s\n", value);
}
```

Returns:

The value of the intent on success; NULL on failure (the intent wasn't found).

asrm_get_locale()

Get the current locale.

Synopsis:

```
#include "asr/asrm.h"

const char* asrm_get_locale()
```

Arguments:

Library:

`libasr`

Description:

The *asrm_get_locale()* funtion retrieves the current local from the global configuration tree.

Returns:

The name of the current locale.

asrm_get_utterance()

Capture an utterance.

Synopsis:

```
#include "asr/asrm.h"

int asrm_get_utterance(asr_module_hdl_t *mod, asr_audio_info_t *audio_info)
```

Arguments:

mod

A pointer to the module handle (optional).

audio_info

The structure in which to store the utterance and set the properties.

Library:

`libasr`

Description:

The *asrm_get_utterance*() function stores an audio sample in the buffer referenced by the *audio_info* parameter by invoking the *asra_get_utterance()* function.

Returns:

0 Success.

-1 An error occurred.

asrm_is_cancellation_request()

Determine whether the current result is a cancellation request.

Synopsis:

```
#include "asr/asrm.h"

int asrm_is_cancellation_request(asr_result_t *result, cfg_item_t
*opt_cancel_section, asr_result_action_t *ret)
```

Arguments:

result

A pointer to the result structure.

opt_cancel_section

A pointer to the cancel configuration section.

ret

The action to take.

Library:

libasr

Description:

The *asrm_is_cancellation_request*() function determines whether the specified result is a cancellation request. If it is, the function sets *ret* to the appropriate action code.

Returns:

1 The result is a cancellation request. The *res* parameter is set to ASR_RECOGNITION_CANCEL.

0 The result is not a cancellation request.

asrm_is_confirmation()

Determine whether the current result is a confirmation.

Synopsis:

```
#include "asr/asrm.h"

int asrm_is_confirmation(asr_result_t *result, cfg_item_t *opt_confirm)
```

Arguments:

result

A pointer to the result.

opt_confirm

A pointer to the configuration item that specifies confirmation options (e.g., "yes", "yeah", "no", "nope", and so on).

Library:

libasr

Description:

The *asrm_is_help_or_cancel*() function determines whether the specified result is a confirmation, either affirmative or negative.

Returns:

1 The response was affirmative.

-1 The response was negative.

0 The response is not understood (neither affirmative nor negative).

asrm_is_help_or_cancel()

Determine whether the current result is either a help request or a cancel request.

Synopsis:

```
#include "asr/asrm.h"

int asrm_is_help_or_cancel(asr_result_t *result, cfg_item_t *opt_help_section,
 cfg_item_t *opt_cancel_section, asr_result_action_t *ret, int perform)
```

Arguments:

result

A pointer to the result structure.

opt_help_section

A pointer to the help configuration section.

opt_cancel_section

A pointer to the cancel configuration section.

ret

The action to take.

perform

A flag to indicate whether to take action on the result.

Library:

`libasr`

Description:

The *asrm_is_help_or_cancel*() function determines whether the specified result is either a help request or a cancel request. If it is one of these, the function sets *ret* to the appropriate action code. If the result is a help request and *perform* is nonzero, the *asrp_active_help()* function is invoked.

Returns:

1 The result is either a help request or a cancel request. The *res* parameter is set to either ASR_RECOGNITION_CANCEL or ASR_RECOGNITION_RESTART.

0 The result is neither a help request nor a cancel request.

asrm_is_help_request()

Determine whether the current result is a help request.

Synopsis:

```
#include "asr/asrm.h"

int asrm_is_help_request(asr_result_t *result, cfg_item_t *opt_help_section,
asr_result_action_t *ret, int perform)
```

Arguments:

result

A pointer to the result structure.

opt_help_section

A pointer to the help configuration section.

ret

The action to take.

perform

A flag to indicate whether to take action on the result.

Library:

libasr

Description:

The *asrm_is_help_request*() function determines whether the specified result is a help request. If it is, the function sets *ret* to the appropriate action code. If *perform* is nonzero, the *asrp_active_help()* function is invoked.

Returns:

1 The result is a cancellation request. The *res* parameter is set to `ASR_RECOGNITION_RESTART`.

0 The result is not a cancellation request.

asrm_next_module()

Find the next module.

Synopsis:

```
#include "asr/asrm.h"

asr_module_hdl_t* asrm_next_module(asr_module_hdl_t *module)
```

Arguments:

module

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_next_module*() function returns the module pointed to by the specifed module's "next" pointer.

Returns:

On success, a pointer to the handle for the module following the specified module; otherwise, a pointer to the module list.

asrm_post_result()

Post a recognition result.

Synopsis:

```
#include "asr/asrm.h"

asr_result_action_t asrm_post_result(asr_result_t *result)
```

Arguments:

result

A pointer to the result to post.

Library:

`libasr`

Description:

The *asrm_post_result*() function posts recognition results to the current conversation.

Returns:

The action to take on the result.

asrm_recognizer_hold()

Place a hold on the recognizer.

Synopsis:

```
#include "asr/asrm.h"

int asrm_recognizer_hold(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle of the recognizer.

Library:

`libasr`

Description:

The *asrm_recognizer_hold*() function stops the current recognition turn and increments the recognizer hold count. The recognizer stops acquiring audio and processing results for the current request.

Returns:

The number of holds.

asrm_recognizer_release()

Release a hold on the recognizer.

Synopsis:

```
#include "asr/asrm.h"

int asrm_recognizer_release(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle of the recognizer.

Library:

`libasr`

Description:

The *asrm_recognizer_release*() function reduces the hold count on the recognizer by one. If no holds remain, it starts the recognizer.

Returns:

The number of holds remaining.

asrm_recognizer_start()

Start a recognition request.

Synopsis:

```
#include "asr/asrm.h"

int asrm_recognizer_start(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_recognizer_start*() function starts a recognition request by invoking *asr_start()*.

Returns:

0 Success.

<0 An error occurred.

asrm_recognizer_stop()

Stop a recognition request.

Synopsis:

```
#include "asr/asrm.h"

int asrm_recognizer_stop(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_recognizer_stop*() function stops a recognition request by invoking *asr_stop()*.

Returns:

0 Success.

<0 An error occurred.

asrm_set_active_sections()

Set the specified configuration as active.

Synopsis:

```
#include "asr/asrm.h"

int asrm_set_active_sections(asr_module_hdl_t *mod, int num_sections, const
char **sections)
```

Arguments:

mod

A pointer to the module handle.

num_sections

The number of configuration sections.

sections

A pointer to the array of configuration sections.

Library:

`libasr`

Description:

The *asrm_set_active_sections*() function sets the vendor configuration sections used by the module. This allows different grammars to be used by the NLAL in different circumstances.

Returns:

0 Success.

-1 An error occurred.

asrm_set_exclusive()

Set a module as exclusive.

Synopsis:

```
#include "asr/asrm.h"

int asrm_set_exclusive(asr_module_hdl_t *mod, asr_context_hdl_t *context)
```

Arguments:

mod

A pointer to the module handle.

context

A pointer to the context handle.

Library:

`libasr`

Description:

The *asrm_set_exclusive*() function sets the specified module as exclusive. Only the exclusive module will see results from recognition sessions until its exclusive status is removed. Exclusive modules must implement the *select_result*() and *on_result*() callback functions; otherwise, the call to *asrm_set_exclusive*() will fail. If a recognition context is provided, it will be used instead of the contexts described in the active configuration sections.

Returns:

0 Success.

-1 An error occurred.

asrm_set_locale()

Set the locale in the global configuration.

Synopsis:

```
#include "asr/asrm.h"

int void asrm_set_locale(const char *locale)
```

Arguments:

locale

The name of the locale to set.

Library:

`libasr`

Description:

The *asrm_set_locale*() function sets the locale in the global configuration tree to the specified value.

Returns:

Nothing.

asrm_set_utterance()

Copy an utterance to the specified buffer.

Synopsis:

```
#include "asr/asrm.h"

int asrm_set_utterance(asr_module_hdl_t *mod, asr_audio_info_t *audio_info,
uint32_t ms_offset)
```

Arguments:

mod

A pointer to the module handle (for error logging).

audio_info

A pointer to the structure in which to store the utterance.

ms_offset

The offset (in milliseconds) of the utterance.

Library:

`libasr`

Description:

The *asrm_set_utterance*() function copies the last captured audio sample to the buffer referenced by the *audio_info* parameter, at the offset specified by the *ms_offset* parameter. If the requested offset results in a buffer overrun, an error is returned. If the audio capture has not completed, an error is returned.

Returns:

0 Success.

EBUSY Capture has not completed.

ENOMEM Buffer overrun or other memory error.

asrm_slog()

Capture logging information for the module.

Synopsis:

```
#include "asr/asrm.h"

int asrm_slog(asr_module_hdl_t *mod, int severity, const char *fmt,...)
__attribute__((format(printf
```

Arguments:

mod

The module handle.

severity

The severity of the condition that triggered the message. For more information on severity levels, see *slogf()* in the *QNX C Library Reference*. Valid values include:

- `_SLOG_INFO`
- `_SLOG_WARN`
- `_SLOG_ERROR`
- `_SLOG_CRITICAL`

fmt

The format string to print to the log buffer. This may include tokens that will be replaced by values of variable arguments appended to the end of the call. The max length of an expanded log message is 1024 characters (this includes all format substitutions and the null terminator).

Library:

`libasr`

Description:

The *asrm_slog*() function sends debugging information to the appropriate log. Log messages will be written to the log buffer only if their severity is greater than or equal to the specified severity.

Returns:

0 Success.

<0 An error occurred.

asrm_strdup_result()

Copy the text from a result.

Synopsis:

```
#include "asr/asrm.h"

char* asrm_strdup_result(asr_result_t *result, unsigned int index)
```

Arguments:

result

A pointer to the result to copy from.

index

The index to begin copying from.

Library:

`libasr`

Description:

The *asrm_strdup_result*() method copies the terminals from the specified result to a new string, starting from the specified index.

Returns:

asrm_unset_exclusive()

Remove a module's exclusive setting.

Synopsis:

```
#include "asr/asrm.h"

void asrm_unset_exclusive(asr_module_hdl_t *mod)
```

Arguments:

mod

A pointer to the module handle.

Library:

`libasr`

Description:

The *asrm_unset_exclusive*() function removes the exclusive setting of the specified module.

Returns:

Nothing.

asrnl_check_section_rules()

Check the active configuration for BNF rules.

Synopsis:

```
#include "asr/asrm.h"

int asrnl_check_section_rules(cfg_item_t *base, char **match_beg, char
**remaining_utt, cfg_item_t **match_item, cfg_item_t *payload)
```

Arguments:

base

A pointer to the base of the configuration tree.

match_beg

A pointer to the start of the matching configuration section.

remaining_utt

A pointer to the remainder of the utterance.

match_item

The configuration item that describes the BNF rule that the NLAL matched the utterance against.

payload

A pointer to the configuration item to check.

Library:

`libasr`

Description:

The *asrnl_check_section_rules*() checks the active configuration section for the BNF rules that the NLAL used to extract intents.

Returns:

The confidence level of the result.

asrnl_evaluate_result()

Evaluate a result against the configuration.

Synopsis:

```
#include "asr/asrm.h"

asr_result_t* asrnl_evaluate_result(asr_result_t *result)
```

Arguments:

result

A pointer to the result to evaluated.

Library:

`libasr`

Description:

The *asrnl_evaluate_result*() function checks the recognized speech in the specified result against the active configuration sections to interpret it as a rule or an intent. If the result is successfully interpreted, it is appended to the results list.

Returns:

A pointer to the updated results list.

asrv_get_common_value()

Get the value for a specified key.

Synopsis:

```
#include "asr/asrm.h"

const char* asrv_get_common_value(const char *key)
```

Arguments:

key

The key string to search on (e.g., "locale").

Library:

`libasr`

Description:

The *asrv_get_common_value*() function finds the value of the configuration item corresponding to the specified key and that matches at least one other configuration item's value.

Returns:

The value of the configuration item; NULL if it isn't found.

find_phrase()

Return the ID of a configuration item containing a specified string.

Synopsis:

```
#include "asr/asrm.h"

int find_phrase(cfg_item_t *base, const char *result, int start_terminal, int
 *terminal, int *conf, int def_id)
```

Arguments:

base

The configuration structure to search.

result

The result string to search for.

start_terminal

Not used.

terminal

On success, *terminal* is set to the index of the first terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

def_id

A default ID to return if the phrase isn't found.

Library:

`libasr`

Description:

The *find_phrase*() function finds the specified string in the specified configuration and returns the ID of the matching configuration item. The configuration item must contain an exact match.

Returns:

The configuration ID of the matching configuration item; the default ID if a match isn't found.

find_result_phrase()

Return the ID of a configuration item containing a speech result.

Synopsis:

```
#include "asr/asrm.h"

int find_result_phrase(cfg_item_t *base, asr_result_t *result, int
start_terminal, int *terminal, int *conf, int def_id)
```

Arguments:

base

The configuration structure to search.

result

The speech result containing the string to search for.

start_terminal

Not used.

terminal

On success, *terminal* is set to the index of the first terminal in the match.

conf

On success, *conf* is set to the confidence level of the match.

def_id

A default ID to return if the phrase isn't found.

Library:

`libasr`

Description:

The *find_result_phrase*() function finds the specified result phrase in the specified configuration and returns the ID of the matching configuration item. The configuration item must contain an exact match.

Returns:

The configuration ID of the matching configuration item or the default ID if a match isn't found.

strconfstr()

Calculate the confidence that two strings match (case insensitive)

Synopsis:

```
#include "asr/asrm.h"
```

```
int strconfstr(char const *string, int opt_len, char const *find, char
**ret_beg, char **ret_end)
```

Arguments:

string

The larger string to search within.

find

The string to search for.

opt_len

The number of characters starting from *string* to include in the search.

ret_beg

If provided, is set to the character in *string* at the beginning of the matched range.

ret_end

If provided, is set to first the character in *string* past the matched range.

Library:

`libasr`

Description:

The *strconfstr*() function searches for a substring of *string* that loosely matches the *find* string and then returns the confidence level of the match.

Returns:

The confidence level (0 - 1000) of the best match of *find* in *string*.

asrp.h

Functions and data types for prompts.

The `asrp.h` header file provides functions and data types for rendering prompts.

Data types in *asrp.h*

asr_prompt_interface

The prompt interface.

Synopsis:

```
struct asr_prompt_interface {
    const char * version ;
    asrp_module_hdl_t *(* connect )(const asrp_module_interface_t *pmif,
unsigned pmif_size, void *module_data);
    int(* slog )(asrp_module_hdl_t *mod, int severity, const char *format,...)
 __attribute__((format(printf;
    int(*) asrp_processing_flags_t(* start )(asrp_prompt_info_t *pi);
    void(* stop )(asrp_processing_flags_t prompt_services);
    void(* reset )(void);
    void(* active_help )(void);
    void(* section_help )(cfg_item_t *section);
    asrp_processing_flags_t(* play_tts )(char *fmt,...)
__attribute__((format(printf;
}asr_prompt_interface_t;
```

Data:

const char * version

The version of the module.

asrp_module_hdl_t *(* connect)(const asrp_module_interface_t *pmif, unsigned pmif_size, void *module_data)

The connect() callback function.

int(* slog)(asrp_module_hdl_t *mod, int severity, const char *format,...) __attribute__((format(printf

The logging callback function.

int(*) asrp_processing_flags_t(* start)(asrp_prompt_info_t *pi)

The start() callback function.

void(* stop)(asrp_processing_flags_t prompt_services)

The stop() callback function.

void(* reset)(void)

The reset() callback function.

void(* active_help)(void)

The active_help() callback function.

void(* section_help)(cfg_item_t *section)

The section_help() callback function.

asrp_processing_flags_t(* play_tts)(char *fmt,...) __attribute__((format(printf

The play_tts() callback function.

Library:

`libasr`

Description:

This data structure represents the high-level interface between the prompt module and `io-asr`. The callback functions provide the mechanism for `io-asr` to request actions from the prompt module.

asr_prompt_interface_t

Alias for the prompt interface.

Synopsis:

```
#include <asr/asrp.h>

asr_prompt_interface_t;
```

Library:

`libasr`

Description:

This type is an alias for the prompt interface, `asr_prompt_interface`.

asrp_module_hdl_t

The prompt module handle.

Synopsis:

```
#include <asr/asrp.h>
```

```
typedef struct asrp_module_hdl asrp_module_hdl_t;
```

Library:

`libasr`

Description:

This opaque data structure is used by `io-asr` to manage data it passes to and from the prompt module.

asrp_module_interface

The prompt module interface.

Synopsis:

```
struct asrp_module_interface asrp_module_interface_t {
    const char * name ;
    const char * version ;
    int(* init )(void *module_data, cfg_item_t *asr_config);
    int(* rate )(asrp_prompt_info_t *prompt_info, int *visual_rating, int
*audio_rating, void *module_data);
    int(* unrate )(void *module_data);
    asrp_processing_flags_t(* start )(asrp_prompt_info_t *prompt_info, void
*module_data);
    asrp_processing_flags_t(* stop )(void *module_data);
    void(* step )(asr_step_t step, void *module_data);
};
```

Data:

const char * name

The name of the prompt module.

const char * version

The version of the prompt module.

int(* init)(void *module_data, cfg_item_t *asr_config)

Initialize the prompt module.

This function initializes the prompt module. Among other actions it may take, it should open the PPS control object for monitoring.

Arguments

- `module_data` A pointer to any data the prompt module requires for initialization.
- `asr_config` Configuration settings for the module.

Returns

- `0` on success; `-1` on error

int(* rate)(asrp_prompt_info_t *prompt_info, int *visual_rating, int *audio_rating, void *module_data)

Set a rating for this prompt module.

The *rate*() function sets a rating for the prompt interface handled by this module. Higher quality, interactive interfaces (such as the HMI) generally receive higher ratings than less interactive interfaces (such as the console).

Arguments

- `prompt_info` Identifying information about the prompt.
- `visual_rating` The visual rating is returned in this parameter.
- `audio_rating` The audio rating is returned in this parameter.
- `module_data` Optional data relating to the module.

Returns

- `0` on success; `-1` on error

int(* unrate)(void *module_data)

Remove ratings for this prompt module.

The *unrate*() function removes ratings for this module.

Arguments

- `module_data` Optional data relating to the module.

Returns

- `0` on success; `-1` on error

asrp_processing_flags_t(* start)(asrp_prompt_info_t *prompt_info, void *module_data)

Start the prompt module.

The *start*() function starts the prompt module.

Arguments

- `prompt_info` Information to pass to the prompt module.
- `module_data` Optional data relating to the module.

Returns

- `Flags` to indicate which, if any, prompts were rendered.

asrp_processing_flags_t(* stop)(void *module_data)

Stop the prompt module.

The *stop*() function stops the prompt module.

Arguments

- `module_data` Optional data relating to the module.

Returns

- ASRP_NO_PROMPTS on success.

void(* step)(asr_step_t step, void *module_data)

Handle a step.

The *step*() function handles a step. The action taken depends on the step specified.

Arguments

- `step` The step to handle.
- `module_data` Optional data relating to the module.

Library:

`libasr`

Description:

This structure defines the interface from `io-asr` to the prompt module. The prompt module's constructor function passes this structure to *asrp_connect() (p. 114)*. The `io-asr` service invokes the member callback functions depending on the state of the module.

asrp_module_interface_t

Alias for the prompt module interface.

Synopsis:

```
#include <asr/asrp.h>

typedef struct asrp_module_interface asrp_module_interface_t;
```

Library:

`libasr`

Description:

This type is an alias for the prompt module interface, asrp_module_interface.

asrp_post_step()

Post a recognition step.

Synopsis:

```
#include <asr/asrp.h>

int asrp_post_step(asr_step_t step)
```

Arguments:

step

The step to post.

Library:

`libasr`

Description:

The *asrp_post_step*() function posts the specified step to the active prompt module.

Returns:

0 Success.

asrp_prompt_info

The prompt information type.

Synopsis:

```
struct asrp_prompt_info asrp_prompt_info_t {
    asrp_processing_flags_t prompt_flags ;
    const char * audio_url ;
    const char * disp_url ;
    asrp_visual_dialog_t visual_dialog ;
    cfg_item_t * payload ;
    asrp_response_cb_t response_cb ;
};
```

Data:

asrp_processing_flags_t prompt_flags

The type of prompt.

const char * audio_url

The URL for the audio source for an audio prompt.

Acceptable formats include `file://` and `string://` for text for TTS; `wav://` for an audio file to play.

const char * disp_url

The URL for a noninteractive display (text bubbles and other simple notices).

asrp_visual_dialog_t visual_dialog

The visual dialog to display.

cfg_item_t * payload

Data that can be provided for the caller's consumption.

This member can express arbritrary information, so is useful for providing text to display or audio files to load.

asrp_response_cb_t response_cb

Optional callback from the prompt module (on-screen dialogs defined in the payload).

Library:

`libasr`

Description:

This type represents information required to produce a prompt.

asrp_prompt_info_t

Alias for the prompt information type.

Synopsis:

```
#include <asr/asrp.h>

typedef struct asrp_prompt_info asrp_prompt_info_t;
```

Library:

`libasr`

Description:

This type is an alias for the prompt information type, asrp_prompt_info.

asrp_visual_dialog

The prompt visual dialog.

Synopsis:

```
struct asrp_visual_dialog asrp_visual_dialog_t {
    const char * header ;
    int num_items ;
    const char ** item ;
    const char * footer ;
    const char * cancel_button ;
    asrp_response_cb_t response_cb ;
    void * data ;
};
```

Data:

const char * header

Text to display above the array of items.

int num_items

The number of items in the array.

const char ** item

The items to display.

const char * footer

Text to display below the array of items.

const char * cancel_button

Text to display on the cancel button.

asrp_response_cb_t response_cb

A callback to be invoked on the OK button.

void * data

Data to be passed in the response callback.

Library:

`libasr`

Description:

This type is used to set text to be displayed to the user in a visual dialog in the HMI. The elements of this type represent the header and footer of the dialog, an array of items to display in the center of the dialog, and text to be displayed on the cancel button, as well as a callback function and optional data to pass to the callback.

asrp_visual_dialog_t

Alias for the prompt visual dialog.

Synopsis:

```
#include <asr/asrp.h>

typedef struct asrp_visual_dialog asrp_visual_dialog_t;
```

Library:

`libasr`

Description:

This type is an alias for the prompt visual dialog, asrp_visual_dialog.

Enumerations in *asrp.h*

asrp_processing_flags_t

Prompt type enumeration.

Synopsis:

```
#include <asr/asrp.h>


typedef enum {
      ASRP_NO_PROMPTS == 0
      ASRP_AUDIO_PROMPT == 0x01
      ASRP_VISUAL_PROMPT == 0x02
      ASRP_INTERACTIVE_PROMPT == 0x10
      ASRP_PROMPT_STEP_NOTIFY == 0x20
} asrp_processing_flags_t;
```

Data:

ASRP_NO_PROMPTS

No prompts.

ASRP_AUDIO_PROMPT

Audio prompt.

ASRP_VISUAL_PROMPT

Visual prompt, such as a dialog.

ASRP_INTERACTIVE_PROMPT

Interactive prompt to collect user input.

ASRP_PROMPT_STEP_NOTIFY

The member is used in processing prompts.

Library:

`libasr`

Description:

The `asrp_processing_flags_t` enumeration lists the types of prompts that can be processed. Depending on the stage processing, members of this enumeration could indicate prompts that have been rendered or prompts that are required.

tts_error_class_t

Error class enumeration.

Synopsis:

```
#include <asr/asrp.h>


 enum {
      TSS_ERROR_CLASS_NONE
      TTS_ERROR_CLASS_MODIFIER
      TTS_ERROR_CLASS_URL
      TTS_ERRROR_CLASS_RESOURCE
      TTS_ERROR_CLASS_SYNTHESIS
      TTS_ERROR_CLASS_SYSTEM
} tts_error_class_t;
```

Data:

TSS_ERROR_CLASS_NONE

No error class.

TTS_ERROR_CLASS_MODIFIER

Used with the asrp_set_error() function to provide additional information about the error.

TTS_ERROR_CLASS_URL

The playback URL could not be resolved.

TTS_ERRROR_CLASS_RESOURCE

A required resource was unavailable.

TTS_ERROR_CLASS_SYNTHESIS

There was an error during TTS synthesis.

TTS_ERROR_CLASS_SYSTEM

There was a system-level error.

Library:

`libasr`

Description:

The `tts_error_class_t` enumeration lists the classes of errors that may occur during speech processing. This information can be used to provide information to the user and to populate error logs.

Functions in *asrp.h*

asrp_active_help()

Provide help prompts to the user.

Synopsis:

```
#include <asr/asrp.h>

void asrp_active_help(void)
```

Arguments:

None.

Library:

`libasr`

Description:

The *asrp_active_help*() function provides contextual help prompts to the user.

Returns:

Nothing.

asrp_connect()

Connect to the prompt module.

Synopsis:

```
#include <asr/asrp.h>

asrp_module_hdl_t* asrp_connect(const asrp_module_interface_t *pmif, unsigned
 len, void *data)
```

Arguments:

pmif

The prompt module interface.

len

The size of the prompt module interface.

data

Data associated with the prompt module.

Library:

`libasr`

Description:

The *asrp_connect*() function connects to the prompt module and returns identifying information about the recognizer via the *data* parameter.

Returns:

The prompt module handle.

asrp_get_status()

Get the prompt status.

Synopsis:

```
#include <asr/asrp.h>
```

```
asrp_processing_flags_t asrp_get_status(void)
```

Arguments:

None.

Library:

`libasr`

Description:

The *asrp_get_status*() returns the list of active prompts.

Returns:

Flags to indicate which prompts are active.

asrp_play_item()

Play the audio item at the specified configuration node.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_play_item(cfg_item_t *base, const char *item_path)
```

Arguments:

base

The configuration node to start at.

item_path

A '/' separated list of node names that leads to the required node.

Library:

`libasr`

Description:

The *asrp_play_item*() function resolves the specified configuration and path to a URL and then plays the audio file specified by the URL.

Returns:

Flags to indicate which, if any, prompts were rendered.

asrp_play_tts()

Play the specified text.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_play_tts(const char *fmt,...)
__attribute__((format(printf
```

Arguments:

fmt

The format string for the text. This may include tokens that will be replaced by values of variable arguments appended to the end of the call.

Library:

`libasr`

Description:

The *asrp_play_tts*() function converts the specified text to speech and plays it back. The text may take the form of a string literal or a variable.

Returns:

Flags to indicate which, if any, prompts were rendered.

asrp_play_tts_item()

Play the TTS item at the specified configuration node.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_processing_flags_t asrp_play_tts_item(cfg_item_t
 *base, const char *item_path)
```

Arguments:

base

The configuration node to start at.

item_path

A '/' separated list of node names that leads to the required node.

Library:

`libasr`

Description:

The *asrp_play_tts_item*() function resolves the specified configuration and path to a URL and then plays the TTS item specified by the URL.

Returns:

Flags to indicate which, if any, prompts were rendered.

asrp_play_url()

Play the audio item at the specified URL.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_play_url(const char *url)
```

Arguments:

url

The URL of the resource to play.

Library:

`libasr`

Description:

The *asrp_play_url*() function plays the audio resource specified by the URL.

Returns:

Flags to indicate which, if any, prompts were rendered.

asrp_reset()

Allow prompting after a stop.

Synopsis:

```
#include <asr/asrp.h>

void asrp_reset(void)
```

Arguments:

None.

Library:

`libasr`

Description:

The *asrp_reset*() function resets the prompt control flags to allow prompting again after a call to *asrp_stop()*.

Returns:

Nothing.

asrp_response_cb_t

Prompt response callback function.

Synopsis:

```
#include <asr/asrp.h>

typedef void(* asrp_response_cb_t)(int selection_index, cfg_item_t *payload);
```

Library:

`libasr`

Description:

The *asrp_response_cb_t*() function can be called from a prompt dialog (e.g., when the user clicks OK).

asrp_section_help()

Provide help related to the specified configuration.

Synopsis:

```
#include <asr/asrp.h>

void asrp_section_help(cfg_item_t *base)
```

Arguments:

base

The configuration for the prompt module.

Library:

`libasr`

Description:

The *asrp_section_help*() function plays all the help URLs in the specified configuration.

Returns:

Nothing.

asrp_set_error()

Set error information.

Synopsis:

```
#include <asr/asrp.h>

void asrp_set_error(asrp_module_hdl_t *mod, tts_error_class_t error_class, int
 error, const char *description)
```

Arguments:

mod

The prompt module handle.

error_class

The class of the error that was encountered.

error

An error code. See `/usr/include/errno.h`.

description

A description of the error.

Library:

`libasr`

Description:

The *asrp_set_error*() function writes error information to the log.

Returns:

Nothing.

asrp_slog()

Capture logging information for the prompt module.

Synopsis:

```
#include <asr/asrp.h>

int asrp_slog(asrp_module_hdl_t *mod, int severity, const char *fmt,...)
__attribute__((format(printf
```

Arguments:

mod

The prompt module handle.

severity

The severity of the condition that triggered the message. For more information on severity levels, see *slogf()* in the *QNX C Library Reference*. Valid values include:

- `_SLOG_INFO`
- `_SLOG_WARN`
- `_SLOG_ERROR`
- `_SLOG_CRITICAL`

fmt

The format string to print to the log buffer. This may include tokens that will be replaced by values of variable arguments appended to the end of the call. The max length of an expanded log message is 1024 characters (this includes all format substitutions and the null terminator).

Library:

`libasr`

Description:

The *asrp_slog*() function sends debugging information to the appropriate log. Log messages will be written to the log buffer only if their severity is greater than or equal to the specified severity.

Returns:

0 Success.

<0 An error occurred.

asrp_start()

Request prompt service from registered prompt service providers.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_start(asrp_prompt_info_t *prompt_info)
```

Arguments:

prompt_info

Structure describing the prompt to be played or shown and providing an interactive callback function if required (e.g., for an interactive visual prompt).

Library:

`libasr`

Description:

The *asrp_start*() function renders the prompt specified by *prompt_info*.

Returns:

Flags to indicate which, if any, prompts were rendered. A return value of `0` (ASRP_NO_PROMPTS) indicates that no prompts were rendered.

asrp_stop()

Stop active prompts.

Synopsis:

```
#include <asr/asrp.h>

asrp_processing_flags_t asrp_stop(asrp_processing_flags_t stop_services)
```

Arguments:

stop_services

The prompts to stop (audio and/or video).

Library:

`libasr`

Description:

The *asrp_stop*() function dismisses any visible prompts and stops playing audio prompts.

Returns:

ASRP_NO_PROMPTS on success.

asrv.h

Functions for vendor-specific actions.

The `asrv.h` header file provides functions to interact with vendor-supplied modules.

Functions in *asrv.h*

asrv_audio_acquire_buffer()

Request an audio buffer.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_acquire_buffer(char **buffer, int *bufflen, int *more_data)
```

Arguments:

buffer

The structure to store the audio sample.

bufflen

The size of the buffer.

more_data

An flag to indicate whether more data is available.

Library:

`libasr`

Description:

The *asrv_audio_acquire_buffer*() function requests a buffer.

Returns:

0 Capturing has finished. The buffer is available.

>0 Capturing is ongoing.

<0 An error ocurred.

asrv_audio_close()

Close the audio module.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_close()
```

Arguments:

Library:

`libasr`

Description:

The *asrv_audio_close*() function closes the current audio module.

Returns:

0 Success.

<0 An error occurred.

asrv_audio_open()

Open the audio module.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_open()
```

Arguments:

Library:

`libasr`

Description:

The *asrv_audio_open*() function opens the current audio module.

Returns:

>=0 Success.

<0 An error occurred.

asrv_audio_relinquish_buffer()

Relinquish an audio buffer.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_relinquish_buffer(char *buffer)
```

Arguments:

buffer

The structure that contains the buffer.

Library:

`libasr`

Description:

The *asrv_audio_relinquish_buffer*() function resets the buffer in the *buffer* structure so that it can be used again.

Returns:

0 Success.

-1 An error occurred.

asrv_audio_set_parms()

Set the audio parameters for the module.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_set_parms(int sample_rate, int volume, int frag_size)
```

Arguments:

sample_rate

The audio sample rate.

volume

The audio volume.

frag_size

The audio fragment size.

Library:

`libasr`

Description:

The *asrv_audio_set_params*() function sets the global audio parameters.

Returns:

0 Success.

-1 An error occurred.

asrv_audio_start()

Start the audio module.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_start()
```

Arguments:

Library:

`libasr`

Description:

The *asrv_audio_start*() function causes the module to begin to perform its particular service, for example capturing audio or playing back from a file.

Returns:

0 Success.

-1 An error occurred.

asrv_audio_stop()

Stop the audio capture.

Synopsis:

```
#include "asr/asrv.h"

int asrv_audio_stop()
```

Arguments:

Library:

`libasr`

Description:

The *asrv_audio_stop*() function forces the audio capturing to stop.

Returns:

0 Success.

-1 An error occurred.

asrv_get_active_sections()

Return `io-asr's` active configuration sections.

Synopsis:

```
#include "asr/asrv.h"

int asrv_get_active_sections(char **sections[])
```

Arguments:

sections

A pointer to the sections (the active sections will be returned using this pointer).

Library:

`libasr`

Description:

The *asrv_get_active_sections*() function returns `io-asr`'s active configuration sections.

Returns:

The number of active sections.

asrv_get_common_value()

Get the value for a specified key.

Synopsis:

```
#include "asr/asrm.h"

const char* asrv_get_common_value(const char *key)
```

Arguments:

key

The key string to search on (e.g., "locale").

Library:

`libasr`

Description:

The *asrv_get_common_value*() function finds the value of the configuration item corresponding to the specified key and that matches at least one other configuration item's value.

Returns:

The value of the configuration item; NULL if it isn't found.

asrv_get_context()

Get the active context.

Synopsis:

```
#include "asr/asrv.h"

int asrv_get_context(asr_context_hdl_t **chdl)
```

Arguments:

chdl

The context handle.

Library:

`libasr`

Description:

The *asrv_get_context*() function returns the preloaded context that the current exclusive module has selected to use for recognition.

Returns:

1 on success.

0 if no context is active.

asrv_get_recognizer_sections()

Return `io-asr's` active recognizer configuration sections.

Synopsis:

```
#include "asr/asrv.h"

int asrv_get_recognizer_sections(char *name, cfg_item_t **recognizer_section[])
```

Arguments:

name

This parameter is currently not used.

recognizer_section

A pointer to the sections (the active sections will be returned using this pointer).

Library:

`libasr`

Description:

The *asrv_get_recognizer_sections*() function returns `io-asr`'s active recognizer-related configuration sections.

Returns:

The number of sections returned.

asrv_post_data()

Pass additional data for use with a result.

Synopsis:

```
#include "asr/asrv.h"

asr_result_action_t asrv_post_data(void *hdl, void *data, int error)
```

Arguments:

hdl

The recognizer handle.

data

The data or parameters to be passed to the module.

error

An error code. The error value is currently specific to the ASR vendor used with `io-asr`.

Library:

`libasr`

Description:

The *asrv_post_data*() function specifies additional parameters or passes additional data that the active module requires (i.e., not recognition results). For example, the module may require a vendor-specific data format (e.g., a tracklist generated from a `find music` command).

Returns:

The next action to take; NULL on error. See `asr_result_action_t` for the list of actions.

asrv_post_result()

Handle results from recognizer.

Synopsis:

```
#include "asr/asrv.h"

asr_result_action_t asrv_post_result(void *hdl, asr_result_t *results)
```

Arguments:

hdl

The recognizer handle.

results

The results to post.

Library:

`libasr`

Description:

The *asrv_post_result*() function handles the specified recognition result. It ensures ASR isn't on hold, selects the appropriate module, passes the result to the module for actioning (the module's *on_result*() callback function is invoked), and returns the action to take next.

Returns:

The next action to take.

asrv_post_step()

Post a recognition step.

Synopsis:

```
#include "asr/asrv.h"

void asrv_post_step(asr_step_t step)
```

Arguments:

step

The step to post.

Library:

`libasr`

Description:

The *asrv_post_step*() function handles the specified step. In the case of an active recognition turn it invokes the appropriate module's *step*() callback function.

Returns:

Nothing.

cfg.h

Data types and functions for setting up module configurations.

The `cfg.h` header file contains data types and functions for building and searching a configuration tree. A configuration tree consists of configuration items. These items carry configuration values that are read from configuration files.

Data types in *cfg.h*

cfg_encoder

The configuration encoder.

Synopsis:

```
struct cfg_encoder cfg_encoder_t {
    cfg_item_t * base ;
    cfg_item_t * container ;
};
```

Data:

cfg_item_t * base

The base configuration item, which can be used as the root of a new configuration tree.

cfg_item_t * container

The container configuration item, which sometimes becomes a child node of a base and sometimes is used to construct new configuration items for other uses.

Library:

`libasr`

Description:

The configuration encoder is an interim structure used to encode configurations.

cfg_encoder_t

Alias for the configuration encoder.

Synopsis:

```
#include <asr/cfg.h>
```

```
typedef struct cfg_encoder cfg_encoder_t;
```

Library:

`libasr`

Description:

This type is an alias for the configuration encoder, cfg_encoder.

cfg_item_t

A configuration item.

Synopsis:

```
#include <asr/cfg.h>

typedef struct cfg_item cfg_item_t;
```

Library:

`libasr`

Description:

This opaque type represtents a configuration item. Configuration items are linked together to form a configuration tree, which represents the contents of a configuration file (e.g., `/etc/asr-car.cfg`). Each configuration item represents a single value from the configuration file.

Functions in *cfg.h*

cfg_add_item()

Add a new node to a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_add_item(cfg_item_t *current, const char *cname, const char
*cvalue)
```

Arguments:

current

A pointer to the node to add the new configuration item to. If NULL, the new configuration item becomes the root of a new configuration tree.

cname

The name for the new node. If NULL, the name "anon" is assigned.

cvalue

The value for the new node. If NULL, an empty string is assigned.

Library:

`libasr`

Description:

The *cfg_add_item*() function creates a new configuration item with the specified name (*cname*) and optional value (*cvalue*) and adds it as a child of the specified node (*current*), behind its sibling nodes. Spaces and quotes are removed from the name and value; any variable references are resolved before the new item is created.

Returns:

A pointer to the new node.

cfg_add_item_string()

Create a new node from a key and add it to a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_add_item_string(cfg_item_t *current, const char *ckey_value)
```

Arguments:

current

A pointer to the node to add the new configuration item to. If NULL, the new configuration item becomes the root of a new configuration tree.

ckey_value

A string consisting of the name for the new node and optionally a value. If NULL, the name "anon" is assigned, with an empty string assigned as the value.

Library:

`libasr`

Description:

The *cfg_add_item_string*() function creates a new a configuration item with the name and optional value specified by *ckey_value*. If *ckey_value* contains a '{' character, it will be removed from the assigned value. For example, the key string "mynode = special {" becomes the key "mynode" and the value "special". The new item is added as a child of the specified node (*current*), behind its sibling nodes.

Returns:

A pointer to the new node.

cfg_attach_item()

Detach an item from its current parent and attach it to another node.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_attach_item(cfg_item_t *parent, cfg_item_t *item, int tail)
```

Arguments:

parent

A pointer to the node to attach the item to.

item

A pointer to the item to attach to *parent*.

tail

Specifies whether the child is prepended (`0`) or appended (`1`) to the parent's list of children.

Library:

`libasr`

Description:

The *cfg_attach_item*() function detaches the specified configuration *item* from its parent and attaches it as a child to the specified node (*parent*). If *parent* is NULL,

this function is equivalent to *cfg_detach_item()*. If *parent* isn't NULL, *tail* specifies whether the child is prepended (`0`) or appended (`1`) to the parent's list of children.

Returns:

A pointer to the item; NULL on error.

cfg_clear_item()

Delete the configuration tree under a node.

Synopsis:

```
#include <asr/cfg.h>

void cfg_clear_item(cfg_item_t *current)
```

Arguments:

current

A pointer to the configuration node. This pointer must not be NULL.

Library:

`libasr`

Description:

The *cfg_clear_item*() function destroys the tree rooted at *current*. It deletes all child nodes under *current* and frees the associated memory. The *current* node is not destroyed.

Returns:

Nothing. No errors are logged.

cfg_clone()

Copy one configuration tree into another.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_clone(cfg_item_t *parent, cfg_item_t *root)
```

Arguments:

parent

A pointer to the configuration tree to add the clone to.

root

A pointer to the configuration tree to copy.

Library:

`libasr`

Description:

The *cfg_clone*() function copies the configuration tree specified by *root* into the configuration tree and at the location specified by *parent*. The structure of the *root* tree is maintained in the cloning operation.

Returns:

A pointer to the parent configuration tree on success, or NULL on error.

cfg_create()

Create a configuration item.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_create(const char *name)
```

Arguments:

name

The name to assign to the new configuration item.

Library:

`libasr`

Description:

The *cfg_create*() function creates a configuration item with the specified name.

Returns:

A pointer to the new configuration item.

cfg_delete_item()

Delete the configuration tree starting at a node.

Synopsis:

```
#include <asr/cfg.h>

void cfg_delete_item(cfg_item_t *current)
```

Arguments:

current

A pointer to the configuration node. This pointer must not be NULL.

Library:

`libasr`

Description:

The *cfg_delete_item*() function destroys the tree rooted at *current*. It deletes the *current* node and all child nodes under it, freeing the associated memory.

Returns:

Nothing. No errors are logged.

cfg_destroy()

Destroy an entire configuration tree.

Synopsis:

```
#include <asr/cfg.h>

void cfg_destroy(cfg_item_t *node)
```

Arguments:

node

A pointer to a node in the configuration tree. This pointer must not be NULL.

Library:

`libasr`

Description:

The *cfg_destroy*() function destroys the tree specified by *node*, which doesn't have to be the root. This function finds the root of the tree, deletes all nodes including the root, and frees the associated memory.

Returns:

Nothing. No errors are logged.

cfg_detach_item()

Detach an item from its current parent.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_detach_item(cfg_item_t *item)
```

Arguments:

item

A pointer to the item to detach.

Library:

`libasr`

Description:

The *cfg_attach_item*() function detaches the specified configuration item (*item*) from its parent and corrects any references in the list of siblings, if any.

Returns:

A pointer to the item.

cfg_dup_resolved_item_value()

Find the specified configuration item and get its resolved values.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_dup_resolved_item_value(const cfg_item_t *item, const char *item_path)
```

Arguments:

item

A pointer to the configuration item to begin the search from.

item_path

The path of the configuration item (e.g., "phone/digit-dialing")

Library:

`libasr`

Description:

The *cfg_dup_resolved_item_value*() function searches for the configuration item named by *item_path* starting at the node *item*. If the item is found, it expands the variable references and returns the resolved value.

Returns:

An allocated string with all the variable references expanded.

cfg_dup_resolved_string()

Get an item's resolved values relative to a node.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_dup_resolved_string(const cfg_item_t *current, char *string)
```

Arguments:

current

A pointer to the node to use to expand variables.

string

The string to resolve.

Library:

`libasr`

Description:

The *cfg_dup_resolved_string*() function expands the variable references in a configuration item relative to the specified node and returns the resolved value.

Returns:

An allocated string with all the variable references expanded.

cfg_encoder_add_int()

Add a configuration item with an integer value to an encoder.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_add_int(cfg_encoder_t *e, const char *name, long long
 num)
```

Arguments:

e

A pointer to the encoder structure to add the new configuration item to.

name

The name of the new configuration item.

num

The integer value of the configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_add_int*() function creates a new configuration item with the specified *name* and integer *value* and adds it as a child of the encoder container (after any other child nodes).

Returns:

A pointer to the new configuration item.

cfg_encoder_add_raw_string()

Add a configuration item with a string value to an encoder.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_add_raw_string(cfg_encoder_t *e, const char *name,
const char *value)
```

Arguments:

e

A pointer to the encoder structure to add the new configuration item to.

name

The name of the new configuration item.

value

The string value of the configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_add_string*() function creates a new configuration item with the specified *name* and string *value* and adds it as a child of the encoder container (after any other child nodes). Spaces and quotes are removed from the name, but the value is used as specified.

Returns:

A pointer to the new configuration item.

cfg_encoder_add_string()

Add a configuration item with a string value to an encoder.

Synopsis:

```
#include <asr/cfg.h>
```

```
cfg_item_t* cfg_encoder_add_string(cfg_encoder_t *e, const char *name, const
char *value)
```

Arguments:

e

A pointer to the encoder structure to add the new configuration item to.

name

The name of the new configuration item.

value

The string value of the configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_add_string*() function creates a new configuration item with the specified *name* and string *value* and adds it as a child of the encoder container (after any other child nodes). Spaces and quotes are removed from the name and value before the configuration item is created.

Returns:

A pointer to the new configuration item.

cfg_encoder_attach()

Attach a configuration item to an existing configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_attach(cfg_encoder_t *e, cfg_item_t *attach, char
*name)
```

Arguments:

e

A pointer to an encoder structure.

attach

A pointer to the configuration node to add the new item to.

name

The name of the new configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_attach*() function creates a new configuration item with the specified *name* and adds it as a child of the specifed node, *attach*, behind any existing child nodes. On return, the base and container of the encoder, *e*, points to the new configuration item.

Returns:

e->base, the base of the specified encoder (which points to the new tree).

cfg_encoder_cleanup()

Clean up an encoder.

Synopsis:

```
#include <asr/cfg.h>

void cfg_encoder_cleanup(cfg_encoder_t *e)
```

Arguments:

e

A pointer to the encoder to clean up.

Library:

`libasr`

Description:

The *cfg_encoder_cleanup*() function deletes the base configuration item of the specified encoder, and also zeroes all memory utilized by the encoder structure.

Returns:

Nothing.

cfg_encoder_end_object()

Finish creating a new configuration item using the specified encoder.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_end_object(cfg_encoder_t *e)
```

Arguments:

e

A pointer to the encoder structure indicating the new configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_end_object*() function returns the parent of the configuration item indicated by *e->container*. If there is no parent, *e->container* is returned.

Returns:

e->container, the container of the specified encoder (which points either to the new configuration item or to its parent).

cfg_encoder_init()

Create a new configuration item.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_init(cfg_encoder_t *e, const char *name)
```

Arguments:

e

A pointer to the encoder that contains the items to construct the tree from.

name

The name of the child item.

Library:

`libasr`

Description:

The *cfg_encoder_init*() function creates a new configuration tree from the specified configuration encoder with the specified name. The encoder base is the root of the new tree. A new configuration item with *name* is added as a child of the root. On return, the base of the encoder points to the root of the new tree; the container points to the new configuration item.

Returns:

e->base, the base of the specified encoder (which points to the new tree).

cfg_encoder_start_object()

Create a new configuration item using the specified encoder.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_encoder_start_object(cfg_encoder_t *e, const char *name, const
 char *value)
```

Arguments:

e

A pointer to the encoder structure.

name

The name of the new configuration item.

value

The value of the new configuration item.

Library:

`libasr`

Description:

The *cfg_encoder_start_object*() function creates a new configuration item with the specified *name* and *value*, and then adds it as a child of the node indicated by the specified encoder's container (*e->container*). If *e->container* is NULL, it becomes a single configuration item.

Returns:

e->container, the container of the specified encoder (which points to the new configuration item).

cfg_find_higher_item()

Find a node higher in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_find_higher_item(const cfg_item_t *current, const char *key)
```

Arguments:

current

A pointer to the node to begin the search in.

key

A '/' separated list of node names to search for.

Library:

`libasr`

Description:

The *cfg_find_higher_item*() function performs a restricted search for a node with *key*, starting with *current* and its sibling nodes, and then moving up to the parent of *current*, the siblings of the parent of *current*, and so on.

Returns:

A pointer to the matching configuration node; NULL if the item wasn't found.

cfg_find_item()

Find a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_find_item(const cfg_item_t *node, const char *cname, int levels)
```

Arguments:

node

A pointer to the node to begin the search in.

cname

A '/' separated list of node names that leads to the desired node.

levels

Indicates how high up the hiearchy the search will go (0 = siblings only; -1 = all the way to the root).

Library:

`libasr`

Description:

The *cfg_find_item*() function performs a restricted hiearchical search starting at *node* for a node with *cname*. For example, if *cname* is "phone/digit-dialing", *cfg_find_item*() searches for the "phone" node within the subtree of *node*, and then searches for the "digit-dialing" node within the "phone" node.

Returns:

A pointer to the matching configuration node; NULL if the item wasn't found.

cfg_find_next_item()

Find a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>
```

```
cfg_item_t* cfg_find_next_item(const cfg_item_t *base, const cfg_item_t
*current, const char *key, int levels)
```

Arguments:

base

A pointer to the parent of the current node (used only if current == NULL)

current

NULL or the result of a previous call to *cfg_find_node()* or *cfg_find_next_node()*.

key

A '/' separated list of node names to search for.

levels

Indicates how high up the hiearchy the search will go (0 = siblings only; -1 = all the way to the root).

Library:

`libasr`

Description:

The *cfg_find_next_item*() function performs a restricted hiearchical search for a node with *cname*. For example, if *key* is "phone/digit-dialing", *cfg_find_next_item*() searches for the "phone" node within the subtree of *node*, and then searches for the "digit-dialing" node within the "phone" node.

Returns:

A pointer to the next node with a matching key; NULL if the item wasn't found.

cfg_find_num()

Return the integer value of a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

long long cfg_find_num(const cfg_item_t *node, const char *key, long long
default_num)
```

Arguments:

node

A pointer to the node to start the search from.

key

A '/' separated list of node names to search for.

default_num

An integer to return if the specified key isn't found.

Library:

`libasr`

Description:

The *cfg_find_num*() function invokes the *cfg_find_item()* function to search a configuration tree for a node that matches *key*, starting at *node*. If a match is found, its integer value is returned.

Returns:

The integer value of the matching configuration node; *default_num* if the item wasn't found.

cfg_find_predefined_item()

Find a node earlier in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_find_predefined_item(const cfg_item_t *base, const char *key,
 int levels)
```

Arguments:

base

A pointer to the node to begin the search in.

key

A '/' separated list of node names to search for.

levels

Indicates how high up the hiearchy the search will go (0 = siblings only; -1 = all the way to the root).

Library:

`libasr`

Description:

The *cfg_find_predefined_item*() function performs a restricted search for a node with *key*, starting with the previous siblings of *base*, and then moving up to the parent of *base*, the previous siblings of the parent of *base*, and so on.

Returns:

A pointer to the matching configuration node; NULL if the item wasn't found.

cfg_find_value()

Return the string value of a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_find_value(const cfg_item_t *node, const char *key)
```

Arguments:

node

A pointer to the node to start the search from.

key

A '/' separated list of node names to search for.

Library:

`libasr`

Description:

The *cfg_find_value*() function invokes the *cfg_find_item()* function to search a configuration tree for a node that matches *key*, starting at *node*. If a match is found, its string value is returned.

Returns:

The string value of the matching configuration node; NULL if the item wasn't found.

cfg_get_explicit_value()

Get the value of the specified item.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_get_explicit_value(const cfg_item_t *item)
```

Arguments:

item

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_get_explicit_value*() function returns the value of the specified configuration item.

Returns:

The value string of the specified item, or NULL if *item* is NULL. If there is no value a pointer to a string terminator ('\0') is returned.

cfg_get_key()

Get the key of the specified configuration item.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_get_key(const cfg_item_t *item)
```

Arguments:

item

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_get_key*() function returns a pointer to the key of the specified configuration item.

Returns:

A pointer to the key string of the specified item; NULL if *item* is NULL.

cfg_get_next_item()

Get a node's next item.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_get_next_item(const cfg_item_t *base, const cfg_item_t *current)
```

Arguments:

base

A pointer to the parent of *current*. Must not be NULL.

current

A pointer to the node whose next node is required. Can be NULL.

Library:

`libasr`

Description:

The *cfg_get_next_item*() function returns the next item of *current* or the first child of *base* if *current* is NULL. Note that the first child of *base* might be NULL.

Returns:

A pointer to either the *current* node's next sibling or the first child of *base*.

cfg_get_num()

Get the integer value of the specified configuration item.

Synopsis:

```
#include <asr/cfg.h>

long long cfg_get_num(const cfg_item_t *item)
```

Arguments:

item

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_get_num*() function returns the integer value of the specified configuration item.

Returns:

The value of the specified configuration item, converted to an integer.

cfg_get_parent()

Get the parent of the specified item.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_get_parent(const cfg_item_t *current)
```

Arguments:

current

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_get_parent*() function returns the parent of the specified configuration item.

Returns:

A pointer to the parent of *current*. NULL if *current* is NULL or has no parent.

cfg_get_value()

Get the value of the specifed configuration item.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_get_value(const cfg_item_t *item)
```

Arguments:

item

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_get_key* function gets the string value of the specified item. If *item* has no value, the key is returned.

Returns:

A pointer to the string value of the specified item; a pointer to the item's key if the item has no value; NULL if *item* is NULL.

cfg_insert_item()

Insert a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_insert_item(cfg_item_t *current, const char *ckey, const char
 *cvalue)
```

Arguments:

current

A pointer to the node to add the new configuration item to. If NULL, the new configuration item becomes the root of a new configuration tree.

ckey

The name for the new node. If NULL, the name "anon" is assigned.

cvalue

The value for the new node. If NULL, an empty string is assigned.

Library:

`libasr`

Description:

The *cfg_insert_item*() function creates a new a configuration item with the specifed name (*ckey*) and optional value (*cvalue*). Spaces and quotes are removed from the name and the value; any variable references are resolved before the new item is created. The new item is added as a child of the specified node (*current*), in front of its sibling nodes.

Returns:

A pointer to the new node.

cfg_insert_raw_item()

Insert a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_insert_raw_item(cfg_item_t *current, const char *ckey, const
char *cvalue)
```

Arguments:

current

The node to add the new configuration item to. If NULL, the new configuration item becomes the root of a new configuration tree.

ckey

The name for the new node. If NULL, the name "anon" is assigned.

cvalue

The value for the new node. If NULL, an empty string is assigned.

Library:

`libasr`

Description:

The *cfg_insert_raw_item*() function creates a new a configuration item with the specifed name (*ckey*) and optional value (*cvalue*). Spaces and quotes are removed from the name, but the value is used as specified. The new item is added as a child of the specified node (*current*), behind its sibling nodes.

Returns:

A pointer to the new node.

cfg_load()

Load a configuration file.

Synopsis:

```
#include <asr/cfg.h>

int cfg_load(cfg_item_t *base, const char *path)
```

Arguments:

base

A pointer to the configuration node (usually the root) to populate.

path

The filepath to the configuration file to load.

Library:

`libasr`

Description:

The *cfg_load*() function populates the specified configuration tree, *base*, with the contents of the configuration file specified by *path*. The *base* configuration tree isn't cleared prior to this operation, so the nodes specified within *path* are merged into *base*. Note that duplicate configuration items are permitted, so loading a configuration file twice will yield double entries.

Returns:

0 Success.

-1 An error occurred.

cfg_merge()

Move configuration nodes from a doner to a new parent node.

Synopsis:

```
#include <asr/cfg.h>

void cfg_merge(cfg_item_t *parent, cfg_item_t *doner)
```

Arguments:

parent

A pointer to the node to attach the item to.

doner

A pointer to the node whose next item will be moved to the new parent.

Library:

`libasr`

Description:

The *cfg_merge*() function removes the *doner* node's next item (along with its children) and attaches it as a child of the specified node (*parent*).

Returns:

Nothing.

cfg_replace_item()

Replace a node in a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_replace_item(cfg_item_t *base, const char *ckey, const char
*new_value)
```

Arguments:

base

A pointer to the root of the configuration tree to search.

ckey

The key of the node to search for.

new_value

The value of the replacement node.

Library:

`libasr`

Description:

The *cfg_replace_item*() function deletes the first node under *base* with the key *ckey*. It then inserts a new node with the key *ckey* and value *new_value* as a child of *base*, in front of its sibling nodes.

Returns:

A pointer to the new configuration item.

cfg_resolve_value()

Get the resolved values of the specified configuration item.

Synopsis:

```
#include <asr/cfg.h>

char* cfg_resolve_value(const cfg_item_t *item)
```

Arguments:

item

A pointer to the configuration item.

Library:

`libasr`

Description:

The *cfg_resolve_value*() function expands the variable references in a configuration item and returns the resolved value. For example, the item `"prompt-dir = $(base-dir)/prompt"` becomes `"prompt-dir = /opt/asr/prompt"`.

Returns:

An allocated string with all the variable references expanded.

cfg_traverse()

Return the next node in a depth-first traversal of a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_traverse(const cfg_item_t *base, const cfg_item_t *current)
```

Arguments:

base

A pointer to the base of the configuration tree.

current

A pointer to the current node of the configuration tree.

Library:

`libasr`

Description:

The *cfg_traverse*() function returns subsequent items in a depth-first traversal of the configuration tree rooted at *base*. To traverse the tree, you call *cfg_traverse*() repeatedly, each time passing the result of the previous call as *current*. *cfg_traverse*() returns

NULL if *current* is equal to *base* (i.e., *base* was returned by the last call), indicating the traversal is complete.

Returns:

A pointer to the next node in the traversal of the configuration tree.

cfg_traverse_items()

Return the next node in a depth-first traversal of a configuration tree.

Synopsis:

```
#include <asr/cfg.h>

cfg_item_t* cfg_traverse_items(const cfg_item_t *base, const cfg_item_t
*current, const char *key)
```

Arguments:

base

A pointer to the base of the configuration tree.

current

A pointer to the current node of the configuration tree.

key

The key value to match during the traversal.

Library:

`libasr`

Description:

The *cfg_traverse_items*() function returns subsequent items that match *key* in a depth-first traversal of the configuration tree rooted at *base*. To traverse the tree, you call *cfg_traverse_items*() repeatedly, each time passing the result of the previous call as *current*. The *cfg_traverse_items*() function returns NULL when the traversal is complete.

Returns:

A pointer to the next matching node in the traversal of the configuration tree.

find_quote()

Find a double quote character in a string.

Synopsis:

```
#include <asr/cfg.h>

char* find_quote(char *start)
```

Arguments:

start

The string to search.

Library:

`libasr`

Description:

The *remove_quotes()* function find the first double quote character in the specified string.

Returns:

A pointer to the double quote character.

remove_quotes()

Remove double quotes from a string.

Synopsis:

```
#include <asr/cfg.h>

char* remove_quotes(char *string)
```

Arguments:

string

The string to remove quotes from.

Library:

`libasr`

Description:

The *remove_quotes()* function removes double-quote characters from the beginning and end of the specified string. For example, `"mystring"` becomes `mystring`.

Returns:

The resulting string.

strip_escapes()

Remove escape characters from a string.

Synopsis:

```
#include <asr/cfg.h>

char* strip_escapes(char *string)
```

Arguments:

string

The string to remove quotes from.

Library:

`libasr`

Description:

The *strip_escapes()* function removes one level of escape characters from the specified string. For example, `"The music you specified can\'t be found"` becomes "The music you specified can't be found".

Returns:

The resulting string.

strip_white()

Strip spaces from the beginning and end of a string.

Synopsis:

```
#include <asr/cfg.h>

char* strip_white(char *buffer)
```

Arguments:

buffer

The string to strip spaces from.

Library:

`libasr`

Description:

The *strip_white()* function removes spaces from the beginning and end of the specified string.

Returns:

The resulting string.

mod_types.h

Data types and functions for the audio, recognition, and conversation modules.

The `mod_types.h` header file provides data type definitions and functions for the audio, recognition, and conversation modules.

Definitions in *mod_types.h*

Preprocessor macro definitions for the mod_types.h header file in the libasr library.

Definitions:

`#define ASR_VERSION "0.9"`

The ASR version.

`#define MAX_SLOT_UPDATE 200;`

The maximum number of slot updates allowed.

`#define MAX_TRANSCRIPTIONS 5;`

The maximum number of transcriptions allowed.

Library:

`libasr`

Data types in *mod_types.h*

asr_audio_info

Audio capture information.

Synopsis:

```
typedef struct asr_audio_info {
    uint8_t * buffer ;
    int buffer_len ;
    int sample_size ;
    int sample_rate ;
    int channels ;
}asr_audio_info_t;
```

Data:

uint8_t * buffer

The buffer to carry the audio samples.

int buffer_len

The length of the buffer.

int sample_size

The number of bits per sample.

int sample_rate

The sample rate.

int channels

The number of channels.

Library:

`libasr`

Description:

This structure carries the audio capture properties.

asr_audio_info_t

Alias for the audio capture information.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_audio_info  asr_audio_info_t;
```

Library:

`libasr`

Description:

This type is an alias for the audio capture information type, `asr_audio_info`.

asr_context_hdl_t

The context handle.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_context_hdl asr_context_hdl_t;
```

Library:

`libasr`

Description:

This type is an alias for the context handle, asr_context_hdl.

asr_conversation_if

Conversation module interface.

Synopsis:

```
struct asr_conversation_if asr_conversation_if_t {
    const char * name ;
    const char * asr_version ;
    int localized ;
    int(* init )(void *module_data, cfg_item_t *asr_config);
    void(* destroy )(void *module_data);
    int(* on_asr_step )(asr_step_t step, void *module_data);
   int(* select_result )(asr_result_t *results, void *module_data, asr_result_t
 **selected_result);
    asr_result_action_t(* on_result )(asr_result_t *result, asr_result_t
*results, void *module_data);
    asr_result_action_t(* on_result_data )(void *result, void *module_data,
int error);
    void(* stop )(void);
};
```

Data:

const char * name

The name of this module.

const char * asr_version

The version of ASR that this module was designed for.

The version number is used to prevent newer, incompatible modules from being used with an older build of ASR.

int localized

Indicates whether localized assets (TTS prompts, commands, etc.) are required.

Set to `0` if no localized assets are needed.

int(* init)(void *module_data, cfg_item_t *asr_config)

Initialize the specified module.

Optional. The `io-asr` service calls *init*() for each registered module upon startup.

Arguments

- `module_data` The data provided to `io-asr` when the module registered itself.
- `asr_config` Configuration information for the converstaion module.

Returns

- `0` Success
- `-1` An error occurred. The `io-asr` service logs the error and exits.

void(* destroy)(void *module_data)

Destroy a module.

Optional. The `io-asr` service calls *destroy*() when shutting down a module that has successfuly initialized via the *init*() function.

Arguments

- `module_data` The data provided to `io-asr` when the module registered itself.

int(* on_asr_step)(asr_step_t step, void *module_data)

Handle a change of state.

Optional. The `io-asr` service calls *on_asr_step*() each time the state of the recognizer changes (see *asr_step_t*).

Arguments

- `step` The current or last event that occurred on the recognizer.
- `module_data` The data provided to *io-asr* when the module registered itself.

Returns

- `0` Success.
- `-1` An error occurred. The `io-asr` service logs the error and exits.

int(* select_result)(asr_result_t *results, void *module_data, asr_result_t **selected_result)

Select a recognition result from active modules.

Optional. The `io-asr` service calls *select_result*() for the current exclusive module if there is one; otherwise, `io-asr` makes the call for all active registered modules. If a result is selected, it is returned via *selected_result*.

Arguments

- `results` A list of results containing the hypotheses from the current recognition.
- `module_data` The data provided to `io-asr` when the module registered itself.
- `selected_result` A pointer to the result most suited for the module. The `io-asr` service treats as valid any result with a return value greater than -1.

Returns

- > -1 if a result is selected; -1 if no result is selected.

asr_result_action_t(* on_result)(asr_result_t *result, asr_result_t *results, void *module_data)

Handle a selected result.

Optional. The `io-asr` service calls *on_result*() for a module only if no other module has a result with a higher confidence level. Results found to be relevant to this module won't be processed if this function isn't defined.

Arguments

- `result` A reference to the selected result within the results list.
- `results` The full list of results.
- `module_data` The data provided to `io-asr` when the module registered itself.

Returns

- The next action to take. See `asr_result_action_t` for the list of actions.

asr_result_action_t(* on_result_data)(void *result, void *module_data, int error)

Pass additional data for use with a result.

Optional. The `io-asr` service calls *on_result_data*() to specify additional parameters or pass additional data that the module requires (i.e., not recognition results). For example, the module may require a vendor-specific data format (e.g., a tracklist generated from a `find music` command).

Arguments

- `module_data` The data or parameters to be passed to the module.
- `error` An error code. The error value is currently specific to the ASR vendor used with `io-asr`.

Returns

- The next action to take; NULL on error. See `asr_result_action_t` for the list of actions.

void(* stop)(void)

Stop the module.

Optional. The `io-asr` service calls *stop*() if the speech session is canceled before an *on_result*() callback has completed. This callback can be useful to break out of any function that blocks for an extended period of time in the *on_result*() callback. A new speech session can't be started until the *on_result*() callback returns.

Library:

`libasr`

Description:

This structure defines the interface from `io-asr` to the conversation modules. Each conversation module's constructor function passes this structure to the *asrm_connect() (p. 68)* function. The `io-asr` service invokes the member callback functions depending on the state of the module.

asr_conversation_if_t

Alias for the conversation module interface.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_conversation_if asr_conversation_if_t;
```

Library:

`libasr`

Description:

This type is an alias for the conversation module interface, asr_conversation_if.

asr_module_hdl_t

The module handle.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_module_hdl asr_module_hdl_t;
```

Library:

`libasr`

Description:

This opaque type represents the module handle. The module handle is used internally by `io-asr` to manage data passed between the modules.

asr_recognizer_hdl_t

An alias for the recognizer handle, `asr_recognizer_hdl`.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_recognizer_hdl asr_recognizer_hdl_t;
```

Library:

`libasr`

Description:

This type is an alias for the opaque recognizer handle, `asr_recognizer_hdl`.

asr_recognizer_if

The recognizer interface.

Synopsis:

```
struct asr_recognizer_if asr_recognizer_if_t {
    const char * name ;
    const char * version ;
    int(* init )(cfg_item_t *config_base);
    int(* cleanup )();
    int(* start )();
    int(* stop )();
    void(* step )(asr_step_t step);
    asr_context_hdl_t *(* context_create )(cfg_item_t *cfg);
    int(* context_save )(asr_context_hdl_t *hdl, cfg_item_t *cfg);
   int(* context_add_entries )(asr_context_hdl_t *hdl, cfg_item_t *cfg, const
 char *slot_identifier, asr_slot_entry_t *slot_entry, int num_slot_entries);
    int(* context_delete_entries )(asr_context_hdl_t *hdl, cfg_item_t *cfg,
const char *slot_identifier, asr_slot_entry_t *slot_entry, int
```

```
num_slot_entries);
    int(* context_destroy )(asr_context_hdl_t *hdl);
    int(* get_utterance )(asr_audio_info_t *audio_info);
    int(* set_utterance )(asr_audio_info_t *audio_info, uint32_t offset_ms);
};
```

Data:

const char * name

The name of the module.

const char * version

The version of ASR that this module was designed for.

The version number is used to prevent newer, incompatible modules from being used with an older build of ASR.

int(* init)(cfg_item_t *config_base)

Initialize the module.

The `io-asr` service calls *init*() for each registered module on startup. The *init*() function sets the recognizer properties. The properties that are required vary by vendor.

Arguments

- `config_base` Configuration data for this recognizer. See *cfg_item_t* (p. 134).

Returns

- `0` Success.
- `<0` An error occurred.

int(* cleanup)()

Clean up memory and data after shutting down a module.

The `io-asr` service calls *cleanup*() after shutting down a module to release any memory, destroy mutexes or condvars, or handle any data that must be changed as a result of the module shutting down. The exact requirements of the cleanup vary by vendor.

Returns

- `0` Success.
- `<0` An error occurred.

int(* start)()

Start the module.

The `io-asr` service calls *start*() to start a recognition request. The recognizer should collect and process the audio sample, and then provide status and results via the API defined in the ASR vendor interface, `asrv.h`. This call must be asynchronous and the recognition operation started must be interuptable via a call to the *stop*() callback.

Returns

- `0` Success.
- `<0` An error occurred.

int(* stop)()

Stop the module.

The `io-asr` service calls *stop*() to stop the current recognition operation. The recognizer stops audio acquisition and stops processing results.

Returns

- `0` Success.
- `<0` An error occurred.

void(* step)(asr_step_t step)

Handle a recognition step.

The `io-asr` service calls *step*() when the module's current step changes. The *step*() function takes the appropriate action depending on what the step is.

Arguments

- `step` The step to handle.

Returns

- `0` Success.
- `<0` An error occurred.

asr_context_hdl_t *(* context_create)(cfg_item_t *cfg)

Create a context.

The `io-asr` service calls *context_create*() during the recognition process to create a recognition context.

Arguments

- `cfg` The configuration structure for the recognizer.

Returns

- A pointer to the new context handle on success.

int(* context_save)(asr_context_hdl_t *hdl, cfg_item_t *cfg)

Save a context.

After `io-asr` has created a context by invoking *context_create*(), it calls *context_save*() to save the context in the recognizer's required format, which varies by vendor.

Arguments

- `hdl` The context handle.
- `cfg` The configuration structure for the recognizer.

Returns

- `0` Success.
- `<0` An error occurred.

int(* context_add_entries)(asr_context_hdl_t *hdl, cfg_item_t *cfg, const char *slot_identifier, asr_slot_entry_t *slot_entry, int num_slot_entries)

Add entries to the specified context.

The `io-asr` service calls *context_add_entries*() to add additional entries to the specified context.

Arguments

- `hdl` A pointer to the context handle.
- `cfg` A pointer to the configuration associated with the context.
- `slot_identifier` A pointer to the slot identifier (the position of the new entry).
- `slot_entry` A pointer to the array of new entries.
- `num_slot_entries` The number of entries to add.

Returns

- `0` Success.
- `<0` An error occurred.

int(* context_delete_entries)(asr_context_hdl_t *hdl, cfg_item_t *cfg, const char *slot_identifier, asr_slot_entry_t *slot_entry, int num_slot_entries)

Delete entries from the specified context.

The `io-asr` service calls *context_delete_entries*() to remove entries from the specified context.

Arguments

- `hdl` A pointer to the context handle.
- `cfg` A pointer to the configuration associated with the context.
- `slot_identifier` A pointer to the slot identifier (the position of the entry).
- `slot_entry` A pointer to the entry.
- `num_slot_entries` The number of entries to delete.

Returns

- `0` Success.
- `<0` An error occurred.

int(* context_destroy)(asr_context_hdl_t *hdl)

Destroy the specified context.

The `io-asr` service calls *context_destroy*() to destroy a context.

Arguments

- `hdl` A pointer to the context handle.

Returns

- `0` Success.
- `<0` An error occurred.

int(* get_utterance)(asr_audio_info_t *audio_info)

Capture an utterance.

The *get_utterance*() function stores an audio sample in the buffer referenced by the *info* parameter. It also sets the associated properties of the utterance:

buffer size, sample size, sample rate, and number of channels. The *get_utterance* function waits until the audio capture has completed before copying the sample and returning.

Arguments

- `audio_info` Indicates the structure in which to store the utterance and set the properties.

Returns

- `0` Success.
- `-1` An error occurred.

int(* set_utterance)(asr_audio_info_t *audio_info, uint32_t offset_ms)

Copy an utterance to the specified buffer.

The *set_utterance*() function copies the last captured audio sample to the buffer referenced by the *info* parameter, at the offset specified by the *offset_ms* parameter. The sample size, sample rate, and number of channels must match the properties of the captured sample. If the requested offset results in a buffer overrun, an error is returned. If the audio capture has not completed, an error is returned.

Arguments

- `audio_info` Indicates the structure in which to store the utterance.
- `offset_ms` The offset (in milliseconds) of the utterance.

Returns

- `0` Success.
- EBUSY Capture has not completed.
- `EINVAL` The audio properties don't match.
- ERANGE Buffer overrun.

Library:

`libasr`

Description:

The recognizer interface provides functions to `io-asr` for managing speech-to-text processing. Each recognizer module's constructor function passes this structure to *asr_connect() (p. 186)*.

asr_recognizer_if_t

An alias for the recognizer interface, `asr_recognizer_if`.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_recognizer_if asr_recognizer_if_t;
```

Library:

`libasr`

Description:

This type is an alias for the recognizer interface, asr_recognizer_if

asr_slot_entry

A transcription slot entry.

Synopsis:

```
typedef struct asr_slot_entry {
    char * word ;
    uint64_t id ;
    asr_transcription_t * transcription ;
    size_t num_transcriptions ;
}asr_slot_entry_t;
```

Data:

char * word

The word buffer.

uint64_t id

The ID of the entry.

asr_transcription_t * transcription

A pointer to the transcriptions for this entry.

size_t num_transcriptions

The number of transcriptions for this entry.

Library:

`libasr`

Description:

A slot entry is used internally by the ASR sub-system to manage context slots. The word buffer contains the terminal string associated with the slot. There are no restrictions on the contents of the word buffer. The `asr_slot_entry_t` structure is a member of the structure *_Entry* (p. 192) stucture in `slot-factory.h`

asr_slot_entry_t

Alias for the transcription slot entry type.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_slot_entry  asr_slot_entry_t;
```

Library:

`libasr`

Description:

This type is an alias for the transcription slot entry type, `asr_slot_entry`.

asr_transcription_s

The transcription type.

Synopsis:

```
typedef struct asr_transcription_s {
    int type ;
    void * data ;
    size_t data_len ;
}asr_transcription_t;
```

Data:

int type

The type of the transcription.

void * data

A pointer to the transcriptions.

size_t data_len

The length of the data.

Library:

`libasr`

Description:

A transcription represents the different ways a context entry word can be spelled. The transcription is associated with a slot entry and is used to update a recognizer context with additional speech information (e.g., names from a phonebook for voice dialing).

asr_transcription_t

Alias for the transcription type.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asr_transcription_s  asr_transcription_t;
```

Library:

`libasr`

Description:

This type is an alias for the transcription type, `asr_transcription_s`.

asra_module_interface

Audio module interface.

Synopsis:

```
struct asra_module_interface asra_module_interface_t {
    const char * name ;
    const char * version ;
    int(* init )(cfg_item_t *asr_config);
    void(* destroy )();
    int(* rate )(const char *url);
    void(* unrate )();
    int(* set_params )();
    int(* open )();
    int(* start )();
    int(* acquire_buffer )(asr_audio_info_t *info, int wait);
    void(* relinquish_buffer )(asr_audio_info_t *info);
    int(* get_utterance )(asr_audio_info_t *info);
    int(* set_utterance )(asr_audio_info_t *info, int offset_ms);
    int(* save_wavefile )(const char *fname);
    int(* stop )();
    int(* close )();
};
```

Data:

const char * name

The name of the module.

const char * version

The version of ASR that this module was designed for.

The version number is used to prevent newer, incompatible modules from being used with an older build of ASR.

int(* init)(cfg_item_t *asr_config)

Initialize the specified module.

The `io-asr` service calls *init*() for each registered module on startup. The *init*() function sets the audio properties.

Arguments

- `asr_config` ASR configuration information (such as the sample rate, number of channels sample size, and so on. The required settings depend on the vendor implementation). See *cfg_item_t* (p. 134).

Returns

- `0` Success.
- `<0` An error occurred.

void(* destroy)()

Destroy a module.

The `io-asr` service calls *destroy*() when shutting down a module that has successfuly initialized via the *init*() function.

int(* rate)(const char *url)

Rate this module for the specified source.

The *rate*() function rates this module's ability to handle the specified audio source URL. The module should rate itself 100 if it can reliably play resources of the specified type, but should supply a lower rating if can't play the resources or if it must perform additional processing first.

Arguments

- `url` The URL for the audio source.

Returns

- The module's rating on success; `-1` on error.

void(* unrate)()

Remove the rating for this module.

The *unrate*() function removes the rating for this audio module.

int(* set_params)()

Set the audio parameters for this module.

The *set_params*() function sets the global audio parameters for this module.

Returns

- `0` Success.
- `-1` An error occurred.

int(* open)()

Open the audio module.

The *open*() function opens this audio module.

Returns

- `1` Success.
- `<1` An error occurred.

int(* start)()

Start the audio module.

The *start*() function causes the module to begin to perform its particular service, for example capturing audio or playing back from a file.

Returns

- `0` Success.
- `-1` An error occurred.

int(* acquire_buffer)(asr_audio_info_t *info, int wait)

Request an audio buffer.

The *acquire_buffer*() function requests a buffer.

Arguments

- `info` The structure to store the audio sample.
- `wait` An optional flag to indicate whether the module should wait for a successful audio sample.

Returns

- `0` Capturing has finished. The buffer is available.
- `>0` Capturing is ongoing.
- `<0` An error ocurred.

void(* relinquish_buffer)(asr_audio_info_t *info)

Relinquish an audio buffer.

The *relinquish_buffer*() function resets the buffer in the *info* structure so that it can be used again.

Arguments

- `info` The structure that contains the buffer.

int(* get_utterance)(asr_audio_info_t *info)

Capture an utterance.

The *get_utterance*() function stores an audio sample in the buffer referenced by the *info* parameter. It also sets the associated properties of the utterance: buffer size, sample size, sample rate, and number of channels. The *get_utterance*() function waits until the audio capture has completed before copying the sample and returning.

Arguments

- `info` Indicates the structure in which to store the utterance and set the properties.

Returns

- `0` Success.
- `-1` An error occurred.

int(* set_utterance)(asr_audio_info_t *info, int offset_ms)

Copy an utterance to the specified buffer.

The *set_utterance*() function copies the last captured audio sample to the buffer referenced by the *info* parameter, at the offset specified by the

offset_ms parameter. The sample size, sample rate, and number of channels must match the properties of the captured sample. If the requested offset results in a buffer overrun, an error is returned. If the audio capture has not completed, an error is returned.

Arguments

- `info` Indicates the structure in which to store the utterance.
- `offset_ms` The offset (in milliseconds) of the utterance.

Returns

- `0` Success.
- `EBUSY` Capture has not completed.
- `EINVAL` The audio properties don't match.
- `ERANGE` Buffer overrun.

int(* save_wavefile)(const char *fname)

Save the captured audio sample as a WAV file.

The *save_wavefile*() copies the captured audio sample as a WAV file with the specified filename.

Arguments

- `fname` The name to use for the WAV file.

Returns

- `0` Success.
- `-1` The file couldn't be opened for writing.

int(* stop)()

Stop the audio capture.

The *stop*() function forces the audio capturing to stop.

Returns

- `0` Success
- `-1` An error occurred.

int(* close)()

Close the audio module.

The *close*() function closes the audio module.

Returns

- `0` on success.

Library:

`libasr`

Description:

This structure defines the interface from `io-asr` to the audio module. Each audio module's constructor function passes this structure to *asra_connect() (p. 56)*.

asra_module_interface_t

Alias for the audio module interface.

Synopsis:

```
#include "asr/mod_types.h"

typedef struct asra_module_interface asra_module_interface_t;
```

Library:

`libasr`

Description:

This type is an alias for the audio module interface, asra_module_interface.

Enumerations in *mod_types.h*

module_status_e

The status of the module.

Synopsis:

```
#include "asr/mod_types.h"


typedef enum module_status_e{
      ASR_MODULE_STATUS_READY
      ASR_MODULE_STATUS_SHUTDOWN
} asr_module_status_t;
```

Data:

ASR_MODULE_STATUS_READY

The module is ready.

ASR_MODULE_STATUS_SHUTDOWN

The module can be shut down.

Library:

`libasr`

Description:

This data type enumerates the states of readiness of modules. Either a module is ready and can interact with `io-asr` or it can be shut down (or unloaded, if implemented as a DLL).

asr_module_status_t

Alias for the module status enumeration.

Synopsis:

```
#include "asr/mod_types.h"

typedef enum module_status_e  asr_module_status_t;
```

Library:

`libasr`

Description:

This type is an alias for the module status enumeration, `module_status_e`.

Functions in *mod_types.h*

asr_connect()

Connect the recognition module to `io-asr`.

Synopsis:

```
#include "asr/mod_types.h"

asr_recognizer_hdl_t* asr_connect(const asr_recognizer_if_t *rif, unsigned
len)
```

Arguments:

rif

The initialized recognizer interface. The name, version, and callbacks in the interface must be set.

len

The size of the recognizer interface.

Library:

`libasr`

Description:

The *asr_connect*() function adds the specified recognizer to `io-asr`'s list of registered modules, and then attaches the specified interface to the new handle.

Returns:

The recognizer handle on success; NULL on error.

stristr()

Finds a matching string (case-insensitive)

Synopsis:

```
#include "asr/mod_types.h"

char* stristr(const char *string, const char *find, int opt_len)
```

Arguments:

string

The string to search within.

find

The string to search for.

opt_len

The maximum length of *string* to search.

Library:

`libasr`

Description:

The *stristr*() function performs a case-insensitive search for *find* in *string*. In other words, this is a case-insensitive version of the standard POSIX *strstr*() function.

Returns:

The search string on success; NULL on error.

protos.h

Functions for module logging.

The `protos.h` header file provides functions for logging.

Functions in *protos.h*

logmsg()

Helper function used to generate log messages.

Synopsis:

```
#include "asr/protos.h"

int logmsg(int severity, const char *fmt,...) __attribute__((format(printf
```

Arguments:

severity

The severity of the condition that triggered the message. For more information on severity levels, see *slogf()* in the *QNX C Library Reference*. Valid values include:

- `_SLOG_INFO`
- `_SLOG_WARN`
- `_SLOG_ERROR`
- `_SLOG_CRITICAL`

fmt

The format string to print to the log buffer. This may include tokens to be replaced by values of variable arguments appended to the end of the call. The max length of an expanded log message is 1024 characters (this includes all format substitutions and the null terminator).

Library:

`libasr`

Description:

The *logmsg*() function sends debugging information with an associated severity to the appropriate log. The log where the data is actually sent is specified by the global variable `log_stdout`. If this variable is nonzero, output generated by this function is printed to the system log.

Log messages are written to the log buffer only if their severity is less than or equal to the current verbosity setting.

NOTE: If the severity of the log message is critical, the program is aborted. If the severity of the log message is _SLOG_ERROR, the program exits with a failure status.

Returns:

0 on success.

-1 on error.

logresult()

Send a recognition result to the log.

Synopsis:

```
#include "asr/protos.h"

int void logresult(const int severity, const asr_result_t *result)
```

Arguments:

severity

The severity of the current result to be logged.

result

A pointer to the recognition result to examine.

Library:

`libasr`

Description:

The *logresult*() function writes details about the specified result to the log. The specified severity must be less than the ASR global severity level.

Returns:

Nothing.

slot-factory.h

Functions and data types for the SlotFactory.

The `slot-factory.h` header file provides functions and data types for interacting with the SlotFactory object, which is used to manage recognition results.

Definitions in *slot-factory.h*

Preprocessor macro definitions for the slot-factory.h header file in the libasr library.

Definitions:

```
#define SlotFactory_DefaultPageSize 4096
```

The default page size.

Library:

`libasr`

Data types in *slot-factory.h*

_Entry

A slot list entry.

Synopsis:

```
struct _Entry SlotFactory_Entry {
    char * word ;
    asr_slot_entry_t * slot ;
    SlotFactory_Entry * next ;
};
```

Data:

char * word

The word association with the entry.

asr_slot_entry_t * slot

The transcription slot entry.

SlotFactory_Entry * next

A pointer to the next slot list entry.

Library:

`libasr`

Description:

This structure holds the word that is associated with the entry, as well as a pointer to the transcription slot entry (see *asr_slot_entry_t* (p. 179)) and a pointer to the next slot entry in the list. This allows entries to be chained in a singly linked list.

_SlotFactory

Structure encapsulating slot factory data.

Synopsis:

```
struct _SlotFactory SlotFactory {
    asr_slot_entry_t * entries ;
    size_t buffer_size ;
    unsigned int page_size ;
    unsigned int max_entries ;
    unsigned int num_entries ;
    SlotFactory_EntryList terminals ;
};
```

Data:

asr_slot_entry_t * entries

A buffer containing an array of transcription slot entries.

size_t buffer_size

The size of the buffer.

unsigned int page_size

The page size.

Optional. Specified at initialization.

unsigned int max_entries

The maximum number of slot entries in the buffer.

unsigned int num_entries

The current number of slot entries in the buffer.

SlotFactory_EntryList terminals

An ordered list of slot entry objects.

Library:

`libasr`

Description:

The slot factory manages an array of ASR slot structures that can be passed to the various ASR context-manipulation functions.

The buffer slot structure is allocated in pages to optimize memory use without frequently reallocating the buffer. The size of a single page of slot entries can be set when the slot factory is initialized. The default page size is 4 KB, which is large enough to hold 168 `asr_slot_entry_t` instances. If a page size is specified when the slot factory is initialized, it is adjusted to the next largest 32-bit aligned buffer size.

SlotFactory_Entry

An alias for the slot list entry type.

Synopsis:

```
#include <asr/slot-factory.h>

typedef struct _Entry SlotFactory_Entry;
```

Library:

`libasr`

Description:

This type is an alias for the slot list entry type, _Entry.

SlotFactory_EntryList

An ordered list of slot entry objects.

Synopsis:

```
#include <asr/slot-factory.h>

typedef SlotFactory_Entry* SlotFactory_EntryList;
```

Library:

`libasr`

Description:

This data type is used to manage ordered lists of slot entry objects. Slot entry lists are in alphabetical order by terminal. Duplicate terminals are not allowed in slot entry lists.

SlotFactory

An alias for the slot factory type.

Synopsis:

```
#include <asr/slot-factory.h>

typedef struct _SlotFactory SlotFactory;
```

Library:

`libasr`

Description:

This type is an alias for the slot factory type, _SlotFactory.

Functions in *slot-factory.h*

SlotFactory_Create()

Create a new slot factory instance.

Synopsis:

```
#include <asr/slot-factory.h>

SlotFactory* SlotFactory_Create(unsigned int num_slots, unsigned int *page_size)
```

Arguments:

num_slots

The initial number of slot entries that the factory's buffer can hold.

page_size

A pointer to a buffer containing the page size used when allocating ASR slot entry buffers. The value obtained from this optional argument is adjusted to the next largest 32-bit aligned value.

Library:

`libasr`

Description:

The *SlotFactory_Create*() function creates a new slot factory instance by allocating an entry buffer that can hold a minimum number of slot entries (specified by *num_slots*). If the *num_slots* argument is zero, the no memory is allocated for the entry buffer and no entry elements are created. These members are updated when the first allocation occurs via the *SlotFactory_createUniqueEntry()* function.

Returns:

A new slot factory instance created on the heap. The ownership of the memory returned by this function is transferred to the calling context, which must delete the instance by calling the *SlotFactory_Delete()* function. If an allocation error occurs, this constructor will return a NULL pointer and set `errno` accordingly.

SlotFactory_createEntry()

Create a new slot entry instance.

Synopsis:

```
#include <asr/slot-factory.h>

SlotFactory_Entry* SlotFactory_createEntry(SlotFactory *self, const char
*terminal, const uint64_t *id)
```

Arguments:

self

A pointer to the slot factory whose slot entry buffer provides the slot entry to associate with the slot entry structure.

terminal

The terminal string to associate with the entry.

id

A pointer to a buffer containing a 64-bit entry ID to associate with the `asr_slot_entry_t` type associated with the returned Entry.

Library:

`libasr`

Description:

The *SlotFactory_createEntry*() function creates a new slot entry instance, associating it with an `asr_slot_entry_t` structure from the factory's buffer.

The *SlotFactory_createEntry*() function creates a new slot entry instance on the heap, copies the specified *terminal* into its word buffer, allocates a new `asr_slot_entry_t` structure from the factory, then associates the two structures. An optional entry ID, *id*, can be specified and is assigned to the newly allocated entry. If no ID is specified, one is assigned automatically.

Returns:

A new instance of the `SlotFactory_Entry` structure, created on the heap, that represents the specified *terminal*. The ownership of the memory returned by this function is transferred to the calling context, which must delete it using the *SlotFactory_Entry_delete()* function.

SlotFactory_createUniqueEntry()

Create a unique Entry instance for the specified terminal.

Synopsis:

```
#include <asr/slot-factory.h>

SlotFactory_Entry* SlotFactory_createUniqueEntry(SlotFactory *self, const char
 *terminal, const uint64_t *id)
```

Arguments:

self

A pointer to the slot factory instance to use to allocate the new slot entry.

terminal

The terminal string to associate with the new slot entry.

id

An optional pointer to a buffer containing a 64-bit entry ID that is associated with the `asr_slot_entry_t` type associated with the returned slot entry. If this argument is NULL, the factory will automatically assign an ID to the entry.

Library:

`libasr`

Description:

If no slot entry for the specified *terminal* has been created by the current factory instance (using the *createUniqueEntry()* function), the *SlotFactory_createUniqueEntry*() function allocates and returns a new slot entry. The new slot entry has an `asr_slot_entry_t` structure associated with it from the slot entry buffer maintained by the factory.

Returns:

A new instance of the `SlotFactory_Entry` type, created on the heap, that represents the specified *terminal*. The ownership of the returned memory is retained by the called context, which will delete it when the Factory is reset. If an Entry for *terminal* has already been created by this factory, its corresponding entry is returned. A NULL pointer is returned if this function fails to create a new entry.

SlotFactory_Delete()

Delete a slot factory instance.

Synopsis:

```
#include <asr/slot-factory.h>

void SlotFactory_Delete(SlotFactory **factory)
```

Arguments:

factory

A pointer to the memory location of a slot factory instance.

Library:

`libasr`

Description:

The *SlotFactory_Delete*() function deletes a slot factory instance that was previously allocated via *SlotFactory_Create()*. This instance is reset (i.e., its buffer is released) and the memory it occupies is deleted. The *factory* pointer is set to NULL.

Returns:

Nothing.

SlotFactory_Entry_Create()

Create a new slot factory entry on the heap.

Synopsis:

```
#include <asr/slot-factory.h>

SlotFactory_Entry* SlotFactory_Entry_Create(const char *terminal)
```

Arguments:

terminal

A pointer to a character buffer containing the terminal to associate with the new entry. The ownership of this memory is retained by the calling context and won't be deleted by the entry structure member functions. If the argument value is NULL, the new entry is created without an associated word buffer.

Library:

`libasr`

Description:

The *SlotFactory_Entry_Create*() function is a constructor for slot entries that, optionally given a terminal string, will create a new entry instance. If no terminal string is provided, the word buffer will remain unallocated. This function is primarily defined for use internally. Use the more robust *SlotFactory_createUniqueEntry()* function instead.

Returns:

A pointer to a newly allocated entry. The ownership of the memory returned by this function is transferred to the calling context, which is responsible for deleting it by calling the *SlotFactory_Entry_delete()* member function.

SlotFactory_Entry_delete()

Delete a slot factory entry.

Synopsis:

```
#include <asr/slot-factory.h>

void SlotFactory_Entry_delete(SlotFactory_Entry **entry)
```

Arguments:

entry

A pointer to the entry to be deleted.

Library:

`libasr`

Description:

The *SlotFactory_Entry_delete*() function deallocates a slot entry that was previously allocated by the *SlotFactory_Entry_Create()* function. This function deletes the *word* buffer associated with the entry; therefore, any external references to this buffer become invalid following a call to this function.

Returns:

Nothing.

SlotFactory_init()

Initialize a slot factory instance.

Synopsis:

```
#include <asr/slot-factory.h>

bool SlotFactory_init(SlotFactory *self, unsigned int num_slots, unsigned int
 *page_size)
```

Arguments:

self

A pointer to the slot factory instance to initialize.

num_slots

The number of slot entries that the initial buffer of the factory can hold.

page_size

A pointer to a buffer containing the page size to use when allocating ASR slot entry buffers. The value obtained from this optional argument is aligned to the next 32-bit aligned value that is large enough to hold the specified page size.

Library:

`libasr`

Description:

The *SlotFactory_init*() function creates a new slot entry buffer large enough to contain *num_slots* entries. If *self* was previously initialized, the memory it manages is leaked. Therefore, this function should not be called twice on the same slot factory instance.

Returns:

True if the factory is initialized; false if an error occurs during initialization. In the error case, `errno` is set to indicate the error that ocurred: either EINVAL if *self* refers to an invalid slot factory instance or ENOMEM if the slot entry buffer cannot be allocated.

SlotFactory_reset()

Reset a slot factory instance.

Synopsis:

```
#include <asr/slot-factory.h>

void SlotFactory_reset(SlotFactory *self)
```

Arguments:

self

A pointer to the slot factory instance to reset.

Library:

`libasr`

Description:

The *SlotFactory_reset*() function releases the slot entry buffer and resets all counters to zero.

Returns:

Nothing.

terminals.h

For internal use only.

The `terminals.h` header is for internal use only.

types.h

Data types for control flow during speech analysis.

The `types.h` header provides data types for the control flow of speech recognition. These data types include result classifications, state enumerations, and error codes.

Definitions in *types.h*

Preprocessor macro definitions for the types.h header file in the libasr library.

Definitions:

```
#define MAX_REC_LEN 140
```

For internal use only.

```
#define MAX_REC_RESULTS 5
```

For internal use only.

```
#define MAX_REC_TERMINAL_LEN 80
```

The maximum length of a terminal in a recognition result.

```
#define MAX_REC_TERMINALS 80
```

The maximum number of terminals in a recognition result.

```
#define isdigit ( __extension__ ({ int _d = (int)(_c); _d = (_d >= '0' && _d
<= '9');}))
```

Determine whether the specified character is a digit.

```
#define isspace ( __extension__ ({ int _d = (int)(_c); _d = (_d == ' ' || _d
== '\f' || _d == '\n' || _d == '\r'  || _d == '\a' || _d == '\b' || _d == '\t'
 || _d == '\v');}))
```

Determine whether the specified character is white space.

```
#define isblank ( __extension__ ({ int _d = (int)(_c); _d = (_d == ' ' || _d
== '\t');}))
```

Determine whether the specified character is a space or a tab.

Library:

`libasr`

Data types in *types.h*

asr_intent

A recognized intent.

Synopsis:

```
typedef struct asr_intent {
    char * field ;
    char * value ;
    asr_result_tag_t tag ;
}asr_intent_t;
```

Data:

char * field

The field of the intent (e.g., search-type).

char * value

The value of the intent (e.g., media).

asr_result_tag_t tag

The result tag for this intent.

Library:

`libasr`

Description:

This data type represents an intent. An intent is an interpreted aim or purpose of an utterance. For example, the intent could be to play a media selection or dial a phone number.

asr_intent_t

Alias for a recognized intent.

Synopsis:

```
#include <asr/types.h>

typedef struct asr_intent  asr_intent_t;
```

Library:

`libasr`

Description:

This type is an alias for the recognized intent type, `asr_intent`.

asr_result

The recognition result.

Synopsis:

```
struct asr_result asr_result_t {
    asr_result_t * next ;
    char * recognizer_id ;
    asr_result_type_t type ;
    asr_result_tag_t tag ;
    char * grammar_name ;
    char * start_rule ;
    char * recognized_speech ;
    int entries ;
    union {
      asr_terminal_t *terminal;
      asr_intent_t *intent;
    }
};
```

Data:

asr_result_t * next

A pointer to the next result.

char * recognizer_id

The ID of the recognizer that generated this result.

asr_result_type_t type

The result type of this result.

asr_result_tag_t tag

The result tag of this result.

char * grammar_name

The name of the grammar used to interpret this result.

This is "dictation" for NL (natural language) recognizers.

char * start_rule

The name of the rule to use to fulfill the user's command.

char * recognized_speech

The string representing the recognized speech.

int entries

The number of entries (either terminal or intent).

anonymous union

asr_terminal_t * terminal

The array of terminals for this result.

asr_intent_t * intent

The array of intents for this result.

Library:

`libasr`

Description:

This type represents the recognition result, which is the text representation of the spoken utterance. The result is either a terminal or an intent.

asr_result_t

Alias for the recognition result.

Synopsis:

```
#include <asr/types.h>

typedef struct asr_result asr_result_t;
```

Library:

`libasr`

Description:

This type is an alias for the recognition result type, asr_result.

asr_result_tag

Properties of the result.

Synopsis:

```
typedef struct asr_result_tag {
    uint64_t id ;
    int confidence ;
    int score ;
    unsigned begin_ms ;
```

```
    unsigned end_ms ;
}asr_result_tag_t;
```

Data:

uint64_t id

The ID of the result.

int confidence

The confidence score for the speech-to-text result.

A higher score indicates greater confidence.

int score

The score is used to determine the correct context for the result.

The lower the score for a context, the better the match.

unsigned begin_ms

The time in the audio capture (in milliseconds) where the terminal begins.

unsigned end_ms

The time in the audio capture (in milliseconds) where the terminal ends.

Library:

`libasr`

Description:

This data type represents properties of a recognition result.

asr_result_tag_t

Alias for the result properties enumeration.

Synopsis:

```
#include <asr/types.h>

typedef struct asr_result_tag  asr_result_tag_t;
```

Library:

`libasr`

Description:

This type is an alias for the result properties enumeration, `asr_result_tag`.

asr_result_type

The status of the result.

Synopsis:

```
typedef struct asr_result_type {
 enum {
  ASR_RECOGNITION_GRAMMAR,
  ASR_RECOGNITION_DICTATION,
  ASR_RECOGNITION_INTENT,
 } recognition_type;
 enum {
  ASR_RESULT_PARTIAL,
  ASR_RESULT_FINAL,
  ASR_RESULT_FAILED,
 } result_type;
 enum {
  ASR_RESULT_OK,
  ASR_RESULT_LOW_CONFIDENCE,
  ASR_RESULT_REJECTED,
  ASR_RESULT_SILENCE,
  ASR_RESULT_CANCELED,
  ASR_RESULT_INTERUPTED,
  ASR_RESULT_MAX_RETRIES,
  ASR_RESULT_ERROR,
 } result_status;
 enum res_error_t {
  ASR_ERROR_NONE,
  ASR_ERROR_REMOTE_SERVER,
  ASR_ERROR_NETWORK,
  ASR_ERROR_GENERAL,
  ASR_ERROR_AUDIO,
  ASR_ERROR_TIMEOUT,
  ASR_ERROR_NO_MEMORY,
  ASR_WARNING_NOTHING_RECOGNIZED,
  ASR_WARNING_SPOKE_TOO_SOON,
  ASR_WARNING_SILENCE,
  ASR_ERROR_SERVICE_UNAVAILABLE,
 } error;
 char *error_description;
} asr_result_type_t;
```

Data:

enum recognition_type

The recognition type. Members include:

ASR_RECOGNITION_GRAMMAR

A grammar result.

ASR_RECOGNITION_DICTATION

A dictation result.

ASR_RECOGNITION_INTENT

An intent result.

enum result_type

The result type. Members include:

ASR_RESULT_PARTIAL

A partial result.

ASR_RESULT_FINAL

A final result

ASR_RESULT_FAILED

A failed result.

enum result_status

The result status. Members include:

ASR_RESULT_OK

The result is OK.

ASR_RESULT_LOW_CONFIDENCE

There is low confidience in the correctness of the result.

ASR_RESULT_REJECTED

The result is rejected.

ASR_RESULT_SILENCE

The recognizer did not find any speech in the audio capture.

ASR_RESULT_CANCELED

The recognition turn was canceled.

ASR_RESULT_INTERUPTED

The recognition turn was interrupted.

ASR_RESULT_MAX_RETRIES

The recognizer has retried the maximum number of times.

ASR_RESULT_ERROR

There was an error with the result.

enum error

Error codes. Members include:

ASR_ERROR_NONE

No error.

ASR_ERROR_REMOTE_SERVER

A server error occurred.

ASR_ERROR_NETWORK

A network error occurred.

ASR_ERROR_GENERAL

A general error occurred.

ASR_ERROR_AUDIO

An audio capture error occurred.

ASR_ERROR_TIMEOUT

A timeout error occurred.

ASR_ERROR_NO_MEMORY

There was insufficient memory for the operation.

ASR_WARNING_NOTHING_RECOGNIZED

The recognizer couldn't detect any speech (possibly because the audio level is too low).

ASR_WARNING_SPOKE_TOO_SOON

The user spoke too soon. This can cause the recognizer to miss the first part of the utterance.

ASR_WARNING_SILENCE

The recognizer couldn't detect any speech.

ASR_ERROR_SERVICE_UNAVAILABLE

The service is temporarily unavailable.

char * error_description

The long description of the error.

Library:

libasr

Description:

This data type describes the status of the recognition result.

asr_result_type_t

The status of the result.

Synopsis:

```
#include <asr/types.h>

typedef struct asr_result_type  asr_result_type_t;
```

Library:

libasr

Description:

This data type describes the status of the recognition result.

asr_terminal

A recognized terminal.

Synopsis:

```
typedef struct asr_terminal {
    char * string ;
    int from_slot ;
    asr_result_tag_t tag ;
}asr_terminal_t;
```

Data:

char * string

The string representation of the recognized word, number, or digit.

int from_slot

The starting slot.

asr_result_tag_t tag

The result tag for this terminal.

Library:

`libasr`

Description:

This data type represents a terminal. A terminal is a recognized block of speech, usually corresponding to a single word or number. In the case of voice dialing, a terminal may correspond to a spoken digit.

asr_terminal_t

Alias for the recognized terminal.

Synopsis:

```
#include <asr/types.h>

typedef struct asr_terminal  asr_terminal_t;
```

Library:

`libasr`

Description:

This type is an alias for the recognized teminal type, `asr_terminal`.

Enumerations in *types.h*

asr_step_e

The recognition step.

Synopsis:

```
#include <asr/types.h>

typedef enum asr_step_e{
      ASR_STEP_LOCALE_CHANGED
      ASR_STEP_SESSION_OPENED
      ASR_STEP_SESSION_ERROR
      ASR_STEP_SESSION_CANCELED
      ASR_STEP_PROMPT_STARTING
      ASR_STEP_PROMPT_STOPPED
      ASR_STEP_RECOGNITION_BEGIN
      ASR_STEP_RECOGNITION_CONFIGURED
      ASR_STEP_PRE_AUDIO_CAPTURE
      ASR_STEP_PRE_SPEECH_SILENCE_TIMEOUT
      ASR_STEP_POST_AUDIO_CAPTURE
      ASR_STEP_LOCAL_QUERY_BEGIN
      ASR_STEP_REMOTE_QUERY_BEGIN
      ASR_STEP_QUERY_END
      ASR_STEP_POSTING_RESULT
      ASR_STEP_REPOSTING_RESULT
      ASR_STEP_RECOGNIZED_SPEECH
      ASR_STEP_UNRECOGNIZED_SPEECH
      ASR_STEP_UNHANDLED_SPEECH
      ASR_STEP_RECOGNIZED_SILENCE
```

```
        ASR_STEP_TURN_COMPLETE
        ASR_STEP_RECOGNITION_END
        ASR_STEP_RECOGNITION_HELD
        ASR_STEP_SESSION_CLOSED
        ASR_STEP_SESSION_CLEANUP
        ASR_STEP_TASK_COMPLETE
        ASR_STEP_SESSION_ABORTED
} asr_step_t;
```

Data:

ASR_STEP_LOCALE_CHANGED

The locale has changed and localized assets have been updated.

ASR_STEP_SESSION_OPENED

A speech session has been opened.

ASR_STEP_SESSION_ERROR

An unrecoverable error has occurred.

This step is be followed by `ASR_STEP_SESSION_CLOSED`.

ASR_STEP_SESSION_CANCELED

The user cancelled the recognition session.

ASR_STEP_PROMPT_STARTING

An audio prompt service has been requested.

The ASR `state` is `prompting`.

ASR_STEP_PROMPT_STOPPED

An audio prompt service has completed.

The ASR `state` is `processing`.

ASR_STEP_RECOGNITION_BEGIN

A recognition turn has started.

ASR_STEP_RECOGNITION_CONFIGURED

The recognition contexts, data, and configuration have been loaded.

ASR_STEP_PRE_AUDIO_CAPTURE

Audio capture is about to start (microphone will be turned on).

The ASR `state` is `listening`.

ASR_STEP_PRE_SPEECH_SILENCE_TIMEOUT

No speech was detected within the silence timeout period.

ASR_STEP_POST_AUDIO_CAPTURE

The microphone has been turned off (either end of speech was detected or asra_stop() was called).

The ASR `state` is `processing`.

ASR_STEP_LOCAL_QUERY_BEGIN

The local ASR service has started processing the captured audio.

ASR_STEP_REMOTE_QUERY_BEGIN

A remote ASR service has started processing the captured audio (expect latency).

ASR_STEP_QUERY_END

The ASR service has returned results.

ASR_STEP_POSTING_RESULT

The recognition result has been generated and is about to be delivered.

ASR_STEP_REPOSTING_RESULT

The recognition result was selected by the wrong module.

ASR_STEP_RECOGNIZED_SPEECH

The module has processed the results.

ASR_STEP_UNRECOGNIZED_SPEECH

The ASR service didn't recognize any speech in the utterance.

ASR_STEP_UNHANDLED_SPEECH

No module selected any of the recognition results.

ASR_STEP_RECOGNIZED_SILENCE

There was no audio captured or the audio was too quiet.

ASR_STEP_TURN_COMPLETE

Result processing is complete.

ASR_STEP_RECOGNITION_END

All module select and result callbacks have completed.

ASR_STEP_RECOGNITION_HELD

The current speech session has been held, to be ended or resumed later.

ASR_STEP_SESSION_CLOSED

The speech session has ended.

ASR_STEP_SESSION_CLEANUP

ASR is cleaning up the speech session.

The ASR `state` is `idle`.

ASR_STEP_TASK_COMPLETE

The user's task has been accomplished (e.g., media playback has begun, phone call has been placed, etc.)

ASR_STEP_SESSION_ABORTED

There was an external cancellation of the recognition session (asr_stop() was called due to a PPS strobe::off message.)

Library:

`libasr`

Description:

This enumeration represents the steps in the recognition process flow. Some of these steps change the state of ASR, as recorded in the `state` attribute of the `/pps/services/asr/control` PPS object. This state change is noted with the enumeration values where applicable.

asr_step_t

Alias for the recognition step enumeration.

Synopsis:

```
#include <asr/types.h>

typedef enum asr_step_e  asr_step_t;
```

Library:

`libasr`

Description:

This type is an alias for the recognition step enumeration, `asr_step_e`.

result_action_e

The result action.

Synopsis:

```
#include <asr/types.h>

typedef enum result_action_e{
      ASR_RECOGNITION_ABORT
      ASR_RECOGNITION_CANCEL
      ASR_RECOGNITION_REPOST
      ASR_RECOGNITION_COMPLETE
      ASR_RECOGNITION_CONTINUE
      ASR_RECOGNITION_RESTART
      ASR_RECOGNITION_REPEAT
      ASR_RECOGNITION_UNKNOWN
      ASR_RECOGNITION_HELD
      ASR_RECOGNITION_HOLD
} asr_result_action_t;
```

Data:

ASR_RECOGNITION_ABORT

Stop the recognition turn as incomplete.

ASR_RECOGNITION_CANCEL

Stop the recognition turn and discard any pending results.

ASR_RECOGNITION_REPOST

Repost the current results.

This allows a conversation module to hand off to other conversation modules.

ASR_RECOGNITION_COMPLETE

The recognition is complete.

Stop the recognition turn if not in continuous mode. Values higher than this indicate a recognizer restart.

ASR_RECOGNITION_CONTINUE

Continue recognizing.

Incremental results are cached.

ASR_RECOGNITION_RESTART

Restart the recognition from the audio source (either get new audio data from the microphone or call asr_set_utterance() to get a new audio buffer).

ASR_RECOGNITION_REPEAT

Restart the recognition from previous recognition features (stored phonemes).

ASR_RECOGNITION_UNKNOWN

The module doesn't understand the command.

This might occur if there has been a context switch.

ASR_RECOGNITION_HELD

A recognition hold has blocked the processing of results.

ASR_RECOGNITION_HOLD

A recognition hold will be triggered, requiring a call to asr_release() or asr_stop().

Library:

`libasr`

Description:

This enumeration represents the actions that can be taken during the processing of recognition results.

asr_result_action_t

Alias for the result action enumeration.

Synopsis:

```
#include <asr/types.h>

typedef enum result_action_e  asr_result_action_t;
```

Library:

`libasr`

Description:

This type is an alias for the result action enumeration, `result_action_e`.

Functions in *types.h*

tolower_m()

Convert the specified character to lowercase.

Synopsis:

```
#include <asr/types.h>

int tolower_m(int c)
```

Arguments:

c

The character to convert.

Library:

`libasr`

Description:

The *tolower_m()* function converts the specified ASCII character to lowercase. If the character isn't in the range of ASCII uppercase characters, it's returned unchanged.

Returns:

The lowercase character on success; the unchanged character on error.

toupper_m()

Convert the specified character to uppercase.

Synopsis:

```
#include <asr/types.h>

int toupper_m(int c)
```

Arguments:

c

The character to convert.

Library:

`libasr`

Description:

The *toupper_m()* function converts the specified ASCII character to uppercase. If the character isn't in the range of ASCII lowercase characters, it's returned unchanged.

Returns:

The uppercase character on success; the unchanged character on error.

Index

U