Synthesiser Access Manager

(Braille Also...)

Design Specification

Revision: 2.05

Last Modified: 26/04/2005

dolphin Oceanic

PO Box 83,
Worcester,
WR3 8TU
England
Tel: +44 01905 754577
Fax: +44 01905 754599
Email: sam@dolphinoceanic.com

1. Overview

The Dolphin Synthesiser Access Manager (SAM) is a client - server application for controlling one or more speech synthesisers and Braille displays.

Its basic purpose is to provide a simple, device independent way for applications such as screen readers to communicate with access devices, and for several applications to share a single device.

1.1 Key Features

The basic features of SAM are as follows.

Operating Systems

SAM is a WIN32 application and is designed for Windows 95. It will also work under Windows NT and future versions of Windows with little or no modification. It will not work under Windows 3.1.

Device Independence

SAM provides complete device independence for the application. Using SAM an application has more control over what is actually spoken by each synthesiser in terms of levels of punctuation and context sensitive exceptions. This ultimately depends on how well a SAM driver is written, but the specification is more detailed than, say, the SSIL.

SAM manages synthesiser installation and configuration and provides the client with a simple list of installed synthesisers and Braille displays to choose from.This will allow the user to change device without re-installing any of their applications.

Extendable

SAM allows each application to modify all possible synthesiser parameters without being restricted to a limited common set. It does this by allowing each device to have a variable number of parameters, which the client can enumerate.

Multi-Lingual

A SAM driver allows full multi-lingual speech (if the device is capable of it). In addition, the client can get a language code number to identify each language. To prevent problems with character mapping and code pages, all text strings passed to and from SAM are in Unicode.

Multiple Client Access

SAM provides a facility to allow several applications to use a single synthesiser or Braille display. Normally this would occur if you had a screen reader and one or more talking applications. (E.g. Hal 95 and An Open Book). There are provisions for complete talking applications and also software that partially talks (where you want to let the screen reader work some of the time).

Modular

SAM is constructed from a number of DLL modules. It will be possible in the future to produce pseudo SAM drivers to interface with SSIL drivers, SAPI, BAPI and other systems or drivers.

Detection

SAM allows each driver to have a detection system, where providing it is not destructive will allow a client application to start talking without having to ever ask the user for the type of their synthesiser.

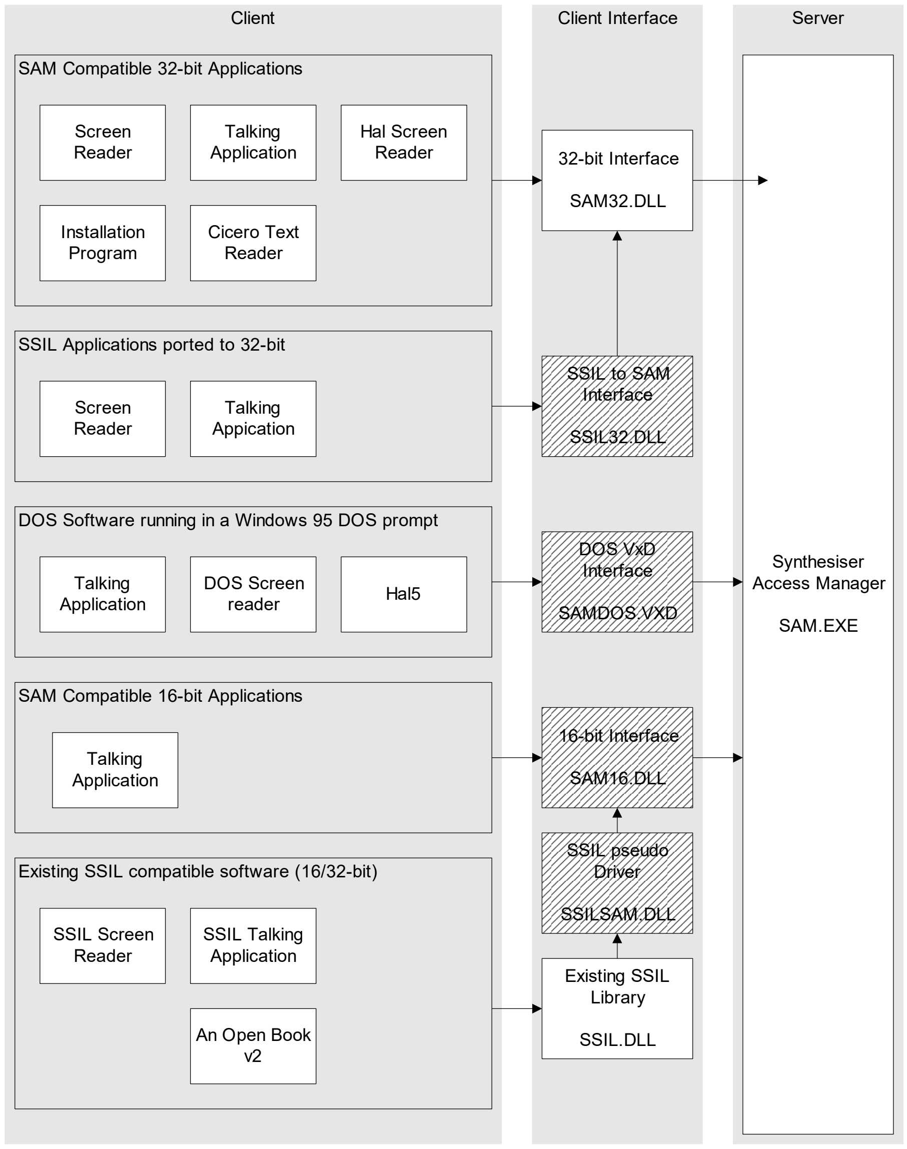
1.2 Clients

SAM supports a number of different types of client software. SAM client software is software that needs to use a speech synthesiser or Braille display. This is either a screen reader, or talking application.

The basic types of client software are as follows (not all these are implemented!).

- SAM compatible 32-bit Applications. These are new WIN32 programs designed for use with SAM.
- SSIL applications ported to 32-bit. These are applications that have been ported to WIN32. SAM will provide a new SSIL32.DLL that contains the same API as the old 16-bit SSIL.DLL, so only minor modifications will have to be made to the speech output code in the existing applications. However all the functionality of SAM will not be available as the application will be restricted by the existing SSIL specification.
- SAM compatible 16-bit Applications. These are applications that run under Windows 95 or NT but are 16-bit. They can use SAM16.DLL that contains an almost identical API to SAM32.DLL. This will make porting from 16 to 32 bit simpler in the future.
- Existing 16-bit SSIL compatible software. This software talks to SSIL.DLL that then talks to a driver. SAM will provide an SSIL compatible driver which talks to SAM instead of accessing a synthesiser directly. This means that any existing SSIL compatible software will be able to use any new SAM device.
- DOS Software. There is no current standard for DOS screen readers. SAM will provide a VxD interface using int 2F to set up a communications channel for DOS software running in a Windows 95 3command prompt.

1.3 Client Interface Diagram



Not all of the modules on the diagram have been implemented by Dolphin. Items shaded with diagonal hatched lines do not yet exist (and may not ever be implemented by Dolphin).

1.4 SAM

SAM is a server application called SAM.EXE. When a client application uses SAM by accessing functions in SAM32.DLL or SAM16.DLL, SAM will automatically be started up by the DLL. SAM can also be started from a shortcut icon to allow for synthesiser configuration and management functions. The client software does not have to concern itself with starting and stopping SAM. This is handled automatically by the interface DLLs.

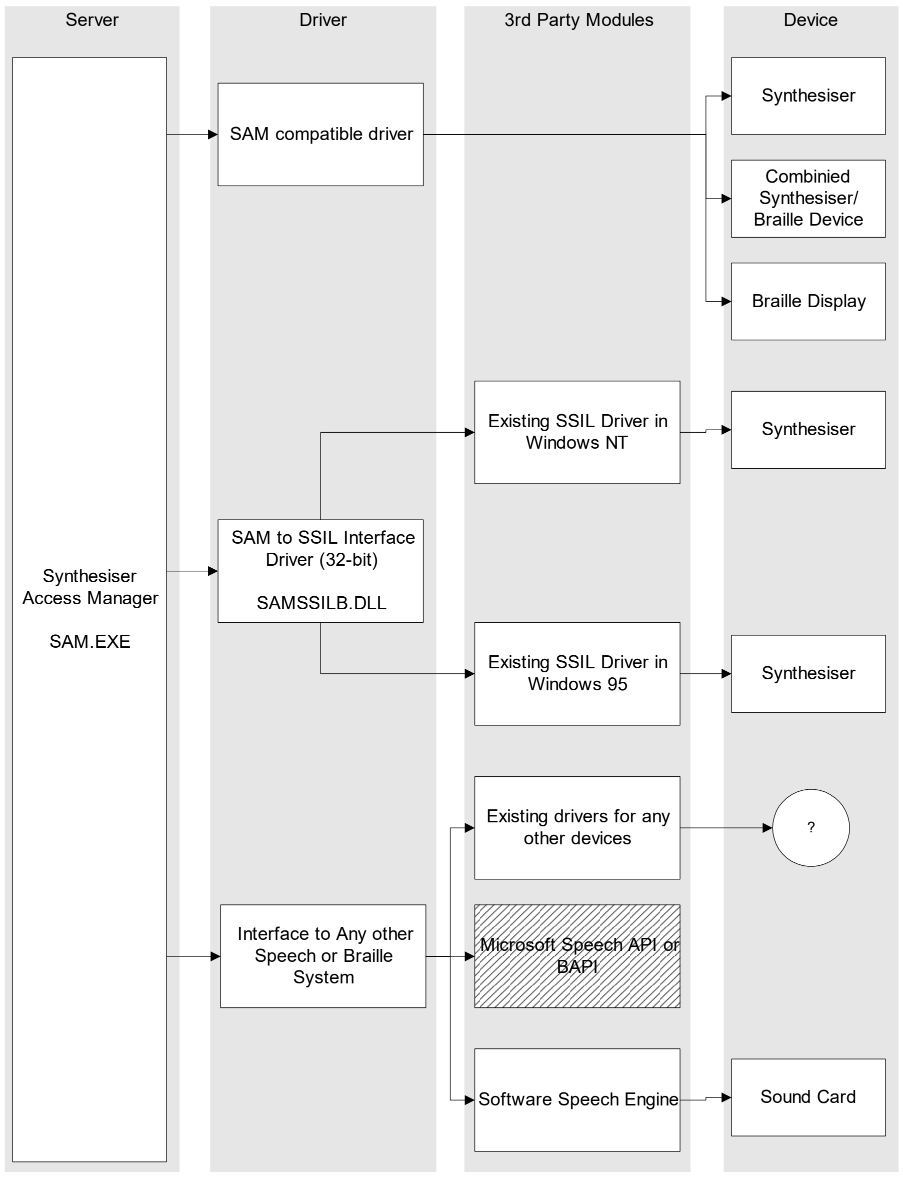
The interface DLLs also performs all the inter-process communications. SAM is a process so has its own address space. All the device drivers are loaded into SAM's process. The interface DLLs set up a private communications channel to SAM.

SAM acts as a server, servicing speech and Braille requests from one or more clients and passing data onto the device drivers.

1.5 Device Drivers

The output from SAM goes through one or more device drivers. Each device driver is a WIN32 DLL with a defined API. The API to a stand-alone driver is the same as the API for the SAM to SSIL interface DLL. SAM does not need to know what type of driver it is talking to, as long as it implements the driver API.

SAM Device Driver Interface Structure


Server
Driver
3rd Party Modules
Device
Synthesiser Access Manager
SAM.EXE
SAM compatible driver
Synthesiser
Combinied Synthesiser/ Braille Device
Braille Display
Existing SSIL Driver in Windows NT
Synthesiser
SAM to SSIL Interface Driver (32-bit)
SAMSSILB.DLL
Existing SSIL Driver in Windows 95
Synthesiser
Existing drivers for any other devices
?
Interface to Any other Speech or Braille System
Microsoft Speech API or BAPI
Software Speech Engine
Sound Card

Not all of the modules on this diagram have been implemented by Dolphin. Items shaded with diagonal hatched lines do not yet exist.

1.6 File Locations

The SAM directory contains SAM.EXE and some of the interface DLL's. However, SAM32.DLL should be installed in the windows system directory so that any client software can load it.

Drivers are stored in sub directories in the SAM directory. Each driver has its own sub directory to prevent clashes of file names. Each driver should be self contained and not require any other files, anywhere else in the system.

1.7 Networks

The basic problem with network software is that there may be different synthesisers on different machines. In addition, a user may log on at any machine and expect to get their preferred settings. SAM is designed to allow installation onto a network drive along with a screen reader.

The location of the SAM directory can be on a network drive so that many users can have access to a common set of drivers. This will also allow driver updates on a network to be simplified. The location of the SAM directory is stored in the registry to allow the interface DLL's to find it.

The configuration for a given machine should be stored by the driver in the registry under HKEY_LOCAL_MACHINE. This will allow network software to use SAM to identify the available synthesisers on whichever machine it has been run.

1.8 Software Speech

It is possible to create a SAM compatible driver that produces software speech through a sound card. In addition, you can use the SAM to SSIL interface to connect to Text Assist (although this is not recommended).

1.9 SSIL Interface

The SSIL interface allows SAM compatible applications to use existing SSIL drivers (and some of them work in NT!). This allows SAM to work with many synthesisers until proper SAM drivers are written. However there are a number of disadvantages to this.

- There is no synthesiser auto-detection facility in SSIL so the user will have to be asked.
- The parameter set is fixed and inflexible (unless you are a DECTalk!).
- Some of the existing drivers are unreliable.
- Synthesiser sharing will not work very well.
- You have less control over punctuation and exceptions.
- You have no control over numbers and context dependent processing.
- You have no control over the placement of index markers.
- It is more difficult to get the synthesiser to say what you want it to.

2. Using SAM

2.1 Interfacing to SAM

Client software interfaces to SAM by using one of SAM's interface DLL's. 32-bit applications should use SAM32.DLL and 16-bit applications should use SAM16.DLL. Applications can dynamically link to one of these DLL's using the supplied LIB files.

SAM is only accessible by using these DLL's.

Note: Only clients using SAM32.DLL will have access to the version 2 API braille functions, 16 bit clients can only use the version 1 functions retained for compatibility.

2.2 Initialisation

When the application starts, it should first call **SamInit**. This will start up SAM if it is not already running and establish a communications channel. An error code will be returned if SAM is not installed or can't be found, or SAM has already been opened by this thread.

When you call **SamInit** you will need to specify if you are a screen reader or a talking application.

A Talking application is one that only talks when it has the input focus. A screen reader is one that tries to make all applications (including itself) speak.

If your application does both then you may need to call **SamInit** twice.

SamInit may fail if you are a screen reader and there is already a screen reader loaded. SAM needs to know what you are in order to sort out sharing of synthesisers.

See Sharing Manager for more information.

The thread that called **SamInit** must be the only thread that calls SAM functions with that handle. The handle is not transferable between threads. SAM identifies the calling thread from its id value.

If you have an application, which is both a screen reader and a talking application (a screen reader with a self-voicing control panel, or a navigation mode), then you should register yourself as a screen reader.

If you want to you can call **SamInit** twice, once as a screen reader and once as a talking application, but each call must be from a different thread.

2.3 Termination

When your application terminates and you have finished with SAM, you should call **SamEnd** for each thread that called **SamInit**. This gives SAM the opportunity to clean up and shut down any devices if necessary.

2.4 Selecting an access device

When SAM has been started, you can call **SamControl(SAMCONTROL_NUM_SYNTH)** to find out how many synthesisers are available or **SamControl(SAMCONTROL_NUM_BRAILLE)** to find out how many Braille displays are available. If these functions return zero then there are no synthesisers (or Braille displays) configured. No further action is possible at this point, as SAM will have already attempted to search for synthesisers and Braille displays. So your application should terminate if it is unable to proceed without speech or Braille.

If the number of units is one you should just use that unit. On a network there may be different devices installed on each workstation.

Otherwise, you should determine which unit to use by comparing the description string for each unit with the description string you saved from the last session. You can get a description string for each available unit by calling **SamQuerySynth** and **SamQueryBraille**.

Your application should provide a method for the user to select an available device. You should create a list from the descriptions and let the user select one.

SAM will query all drivers for any units and build a single list for you. You only have to select a unit from the list. You do not have to deal with driver names.

2.5 Device Capabilities

Once you have picked a synthesiser or Braille display, you can query it for information. **SamQuerySynth** will return a **SYNTHPARAMS** structure that contains information about the number of parameters, languages and capabilities of a synthesiser unit. Similarly, **SamQueryBrailleEx** will return a **BRAILLEPARAMSEX** structure that contains information about the size and capabilities of a Braille display unit. If there is more than one unit available, you may wish to check the device capabilities to help determine which one to use.

2.6 Using a device

You should use **SamOpenSynth** to obtain access to a speech device and **SamOpenBraille** to obtain access to a Braille device. These functions allow you to get a handle to the specified unit. When you have finished using a device, you should close the unit using **SamCloseSynth** (for synthesisers) Or **SamCloseBraille** (for Braille displays). You will have to specify the handle from **SamOpen…** in subsequent calls to SAM to control the device, although you can query a device's capabilities without opening it.

These functions don't really do anything, they allow SAM to know which devices are in use or may be used shortly. SAM will instruct the device driver to shut down the device when no client is accessing it. This is important for battery operated devices, where they can be powered down during inactivity.

2.7 Enumerating speech parameters

Each make of synthesiser has a different set of parameters (pitch, speed, volume, head size etc.) that control its speech. SAM gives you the capability to get a list of available parameters to allow full control of the synthesiser. The important thing is that you don't need to know what the parameter actually does, you can just get a description of it, and a description for each available setting.

Some of the parameters have a known identifier, so you can pick out common parameters such as volume, pitch, language and speed from the list.

The enumeration functions require a unitid number and not a handle. This means that you can enumerate parameters and get descriptions without first opening the unit. This allows you to search for a synthesiser that has a specific feature, such as the ability to speak in a given language.

To enumerate the parameters you first determine how many there are, by checking the **params** value in the **SYNTHPARAMS** structure that was obtained using **SamQuerySynth**.

For each parameter, you call **SamQueryParam** to get information about it.

SamQueryParam takes a unitid, parameter number (numbered from zero) and a pointer to a buffer to receive the data. You should allocate the required memory for the **SAMPARAM** structure.

Each parameter is described by a **SAMPARAM** structure.

```
Struct {
    WORD  type,      // type of parameter
    DWORD range,     // number of values allowed
    long  first,     // offset value for user
    WORD  id,        // parameter id value.
} SAMPARAM;
```

Each parameter has an id value. This value contains an identifier that lets you determine the meaning of the parameter for the more common types (e.g. volume, speed and pitch). Also some other parameters such as punctuation and spelling must be numeric so the client can determine the meaning of some of the parameter values. (E.g. a punctuation value of zero *means* no punctuation).

There are four types of parameters. **Numeric**, *Multi-choice, compound* and *flags*. [Maybe we could add a 'string' type for a parameter to allow the user to enter more complex data. e.g. a filename of an exceptions dictionary]

The type field specifies which type of parameter it is.

Numeric

Numeric values are used where a parameter represents contiguous discrete values, such as speed, pitch or volume.

The numeric values must be contiguous and start at zero. The *first* offset value indicates to the client how to present the value to the user, but internally when the value is stored in the voice block, it should always be numbered from zero.

Multi-Choice

A multi-choice value should be used where there is no sensible numeric value for each parameter. (E.g. "child" or "adult").

Compound

A compound parameter is essentially a multi-choice parameter with a numeric value underneath. The numeric values do not have to be contiguous. This is important for a parameter such as language where different synthesisers may have different supported languages yet you want to be able to save a synthesiser configuration.

Flags

There are other flags defined for each parameter. These give additional information. See **SAMPARAM** for a description of these flags.

2.8 Parameter Descriptions

SAM provides a description for each parameter and a description for each value for each parameter. You can call **SamQueryParamDesc** to get a description for a given parameter and **SamQueryParamChoice** to get a description for the parameter values.

These functions work on all types of parameter, so you can use the same code regardless of the parameter type. For numeric parameters, this may consist of a string containing numeric digits.

All parameter strings that are retrieved are stored in Unicode.

2.9 Braille parameters

Braille displays do not have parameters in the same way as speech synthesisers. Basic information about the Braille unit is contained in the **BRAILLEPARAMSEX** structure obtained by calling **SamQueryBrailleEx**.

Braille displays consist of a number of strips, which can contain braille cells or buttons or both. For example, there may be strips describing a main row of display cells and routing buttons, a row of status and status routing buttons, a row of auxiliary cells, and a number of general-purpose keys. Information on each strip is contained in the **STRIPPARAMS** structure obtained by calling **SamQueryStrip**.

One major distinction between speech synthesisers and Braille displays is that Braille displays can be used to provide input to the application as well as Braille output to the user.

To obtain a description of each general-purpose button or combination of routing button presses on the Braille display you can call **SamQueryButtonStrip**.

All parameter strings that are retrieved are stored in Unicode.

You can get default button combinations for common actions by calling **SamGetButtonCombinationStrip**. You can call **SamIsKeyValidStrip** to determine whether a particular button combination is valid for a particular unit.

2.10 Description Languages

SAM provides the facility for each driver to give descriptions in any language. Descriptions are guaranteed to be available in English, but each driver may support descriptions in other languages as well.

This is important for a multi-lingual product. The user will expect synthesiser parameter descriptions and Braille display button names to be translated, but as the set of possible parameters is variable, only the driver knows what each parameter means, therefore it must also contain translated prompts.

To determine which languages the parameter descriptions are available in you must check the langs variable in the **SYNTHPARAMS** (or **BRAILLEPARAMSEX**) structure. This will be a value of one or higher. This tells you how many different languages the descriptions of parameters and their values are available in.

For each language, you should then call **SamGetLangId** (for speech) or **SamGetBrailleLangId** (for Braille) to retrieve the language identifier number. This is a DWORD that identifies the actual language. There is a set of defined identifiers for most common spoken languages in ***sam.h***. This list does not match the langid codes defined by Microsoft. If you require a language code for a language that is not listed, please contact Dolphin Oceanic Ltd for assistance.

The language code tells you the actual language so you can use the correct prompts in your application.

2.11 The voice block

When anything is spoken, the driver needs to know what voice (or what set of parameters to use) to speak in. Whenever you call SAM with text to speak, you must also pass a pointer to a voice block.

The voice block is a chunk of memory that contains an array of DWORD's, one for each synthesiser parameter.

The size of the voice block in bytes is therefore

```
sizeof(DWORD)*synthparams.params
```

The parameters should be stored in the voice block in the same order as they were enumerated using **SamQueryParam**. The ordering of parameters is therefore entirely up to the driver, but should remain consistent between different sessions. This means that you can save the users settings by storing away a copy of the voice block.

You must pass a complete voice block with each speech call every time. The driver is responsible for optimising output, i.e. only sending the changes to the synthesiser.

If a parameter is changed in mid-sentence and the synthesiser is not capable of performing such a change, the driver will break up the sentence into two. You can change as many parameters in the voice block, as often as you like.

A default voice block can be obtained by calling **SamGetVoice**.

When you call **SamAppend**, your voice block will be validated by SAM and an error will be returned if any of the values are out of range.

You should use the identifier string supplied in the **SYNTHPARAMS** structure to identify your voice block if you store it as part of a users configuration. If a user has more than one synthesiser then you will need to store voice information for each synthesiser as the voice blocks will not be interchangeable between devices. The identifier string indicates whether the voice block you have is compatible with a specific unit.

To prevent the total number of saved voice blocks exploding and to cope with minor differences in the ranges of parameters the identifier string may be identical on two devices that have a slight difference in the range of one or more parameters. (E.g. the user installed an additional language in their synthesiser).

2.12 Preset Voices

SAM provides one or more voices.

A default voice block for the desired spoken language can be retrieved by a call to **SamGetDefaulrVoice**. If you do not intend to provide the user with any voice control, you can just use this voice block without any modification.

Additional preset voices can be retrieved by calling **SamGetVoice,** but the **SAMID_VOICELANG** parameter should be checked to see if it is suitable for the current spoken language. The total number of preset voices available is specified in the **SYNTHPARAMs** structure, in the voices field. Preset voices are numbered from zero to (voices-1*)*.

Each voice parameter in the voice block may be allowed to be set to a 'default' value. This tells the driver to use a default value, which may be based on the setting of the preset parameter (if there is one), or can be internal to the driver. If a parameter can be set to default the **SAMPARAM_DEFAULT** flag will be set in the type field of the **SAMPARAM** structure.

The parameters for volume, speed and language are not allowed to be set to 'default' (the **SAMPARAM** structure for these parameters must not include **SAMPARAM_DEFAULT**).

The numeric value of 'default' is **SAMVAL_DEFAULT** (-1).

When you call **SamGetVoice** the voice block will contain default values for all the parameters that are not flagged as **SAMPARAM_DEFAULT**.

You can call **SamQueryParamChoice** for each preset voice to get its description.

The reason for allowing some parameters to be set to 'default' is for compatibility with SSIL.

2.13 Performing Text to Speech

With a synthesiser, the most vital thing that you want to do is produce speech output.

There are three main functions used for generating and controlling speech output: **SamMute**, **SamAppend,** and **SamSpeak.**

SamMute immediately halts any speech in progress and clears any pending speech. You can then call **SamIndex** to obtain information about where the speech stopped.

SamAppend is used to add words or phrases to the output buffer. This function will not cause the synthesiser to immediately start speaking.

SamSpeak is called to start the speech. All previous text in calls to **SamAppend** will commence to be spoken.

In each call to **SamAppend** you also specify a pointer to a voice block. This allows you to build up sentences with changes in the voice parameters. **SamAppend** accepts a pointer to text and a length. There is no limit to the amount of text that can be sent at one time.

For maximum efficiency, you should call **SamAppend** with the largest chunks of text at a time. Try to avoid calling **SamAppend** with individual characters, unless you need to change the contents of the voice block.

Once the synthesiser is speaking, you can append text to the end of the current speech by calling **SamAppend** with the additional text, and then calling **SamSpeak** to speak it. Providing you don't call **SamMute** all subsequent text will be chained onto the end of the speech.

Try to terminate sentences and phrases with a standard end of sentence character such as a full stop. When you call **SamSpeak** you do not need to terminate the last sentence. Calling **SamSpeak** will do this for you. However, this means that if you then append some more text onto the end, you can't join words together (which is probably a good thing).

SamSpeak implies a phrase ending. Calls to **SamAppend** separated by a call to **SamSpeak** should not be concatenated together into a single sentence, but a phrase ending will be placed between. This provides a method of terminating the phrase without needing to send a carriage return, full stop, etc. In addition, the synthesiser will cleanly finish the phrase without speaking any termination punctuation (e.g. period).

2.14 Querying speech progress

SAM provides indexing facilities so that you can determine the progress of the speech.

The available indexing capabilities may vary between devices so you can check the **SYNTHPARAMS** structure to find out how accurate the information is likely to be. If you request indexing information at a greater resolution than the synthesiser can manage (e.g. you try to index by character, but the synthesiser can only manage word indexing) then the driver will report to the nearest index marker it can.

To perform indexing you should cut up your source text into chunks and call **SamAppend** individually with each chunk. The driver will be able to report back which 'block' is currently being spoken. This means that you decide the actual index positions, by how you cut the text up.

This is why the **SamAppend** function takes a pointer and a length, and not a zero terminated string. Typically, if you want to monitor word progress you will have a continuous string in memory. You can then append each word without having to modify your string in any way. It is also up to you as to how you define what a word is, e.g. if you break up hyphenated words.

When you call **SamAppend**, you also specify a unique index value. It is this value that is passed back when you query the speech progress using **SamIndex**. The index value is a DWORD so you can put whatever values you like in it. E.g. a 32-bit pointer into your original string, a word counter, etc.

For Example, If you want to speak a sentence and know which word is being spoken at any instance, you should call **SamAppend** with each word individually with a unique index value.

To determine the current index position at any given instance you can call **SamIndex**. This function call will return very quickly because the driver will be retrieving the index count from the synthesiser asynchronously in the background.

The call to **SamIndex** will return the index identifier for the currently speaking text block. If the speech has finished, the index identifier returned will be the one specified when you called **SamSpeak**.

As calls to **SamAppend** are concatenated together, you must still put a space between each word!

2.15 Performing Text to Braille

With a Braille display, the most vital thing that you want to do is produce Braille output.

There are three main functions used for generating and controlling Braille output: **SamClearDisplayStrip**, **SamSetDisplayStrip** and **SamSetCursorStrip.**

SamClearDisplayStrip is used to clear all the cells on a display or status strip, and can be used to clear all strips.

SamSetDisplayStrip is used to set the cells of a Braille display strip. The data passed to this function must be of fixed length, as defined by the dimensions in the **STRIPPARAMS** structure.

SamSetCursorStrip is used to set the position, shape and blink rate of a cursor on the display. A cursor may be blinking or static and can appear in any cell in the display area. The cursor may or may not be placed on strips other than the main display, depending on the particular device.

The data passed to **SamSetBrailleStrip** is given in a device independent, language independent form. Each WORD of data defines one Braille cell, as follows:

Bit	Action
0	Dot 1
1	Dot 4
2	Dot 2
3	Dot 5
4	Dot 3
5	Dot 6
6	Dot 7
7	Dot 8
8	Blink dot 1
9	Blink dot 4
10	Blink dot 2
11	Blink dot 5
12	Blink dot 3
13	Blink dot 6
14	Blink dot 7
15	Blink dot 8

This format makes it possible to manipulate cells easily. The dots are arranged in the standard way:

1	●	●	4
2	●	●	5
3	●	●	6
7	●	●	8

When you set the cursor shape (using **SamSetCursorStrip**) the format of the shape data is the same as above, except only the lower byte is used. Typical cursor shapes are dot 8, dots 7 and 8 or all 8 dots.

SAM provides a simple function, **SamTranslateBraille**, to translate a Unicode string into this Braille dot format. However, this function will translate only the basic Latin characters (U+0020 - u+007f) and the translation of each character is fixed as the North American Braille Computer Code (NABCC).

2.16 Unicode

All of the text passed to or from SAM is stored in Unicode, unless otherwise specified. Unicode text strings are indicated in the prototypes by the presence of WCHAR * instead of the normal char *. For non-Microsoft users, a WCHAR is an *unsigned short*.

Unicode is an international standard for representing characters and avoids all of the horrible code page problems of multi-lingual software.

SAM used Unicode version 1.1. The specification is available from the Unicode Consortium.

If your application does not use Unicode then you can use **MultiByteToWideChar** and **WideCharToMultiByte** windows functions to convert to and from Unicode, although you may have trouble with special sound effect characters. Make sure you get the correct code page when you do this!.

Unicode values are usually indicated with the notation U+#### where #### is a four digit hex number, E.g. U+0041 is 'A', U+20A4 is '£'.

2.17 Character Sets

One of the basic problems with speech synthesisers is that different models do different things with the same characters.

SAM provides a more exact specification of what characters each synthesiser should accept and what it should do with them. Using this information, you should be able to arrange for your software to produce the same actual words on any synthesiser device.

Each synthesiser unit provides you with character set information. This defines which characters it will accept to form words.

There are four sets of characters. These are known as the **alphabetic**, **modifier**, **punctuation** and **special** characters.

Only the alphabetic characters will be used to form words. Modifier characters should not produce any speech by themselves but only affect the pronunciation of alphabetic characters.

The punctuation characters will always be spelt by the synthesiser, even if they are in the middle of a word, i.e. they should break up the word.

These three sets of characters must be mutually exclusive.

To obtain the sets of characters you should call **SamGetCharSet** for each language. A different set of characters is allowed per language.

The driver should only take notice of characters in the speech string that are in its supported character set. Any other characters will be ignored. (You shouldn't pass any characters that are not in the allowed character sets.

A typical character set might be:

Alphabetic:
`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789`

Modifier: `,.'-!?`

Punctuation `"£$%^&*()=+][{}#~'@/;:`

Numbers are part of the language because numbers should always be spoken as words, i.e. the synthesiser should form words from sequences of digits in the same way as it forms words from sequences of letters. E.g. 123 should be spoken as one hundred and twenty three. If you want to announce single digits then you can either make sure that they are separated by a space or you can spell them (see Spelling later).

If the device is not capable of speaking number sequences as whole words, then the flag **SAMCAPS_NUMBERS** will not be set in the `synthparams.caps` structure.

The space character (U+0020) must be part of the Modifier characters, because it does not produce any sound by itself.

The alphabetic character set must contain both upper and lower case as both are equivalent when it comes to forming words, however the synthesiser must speak the same thing regardless of upper or lower case.

2.18 Sound Effects

Special characters can be put into the speech string to cause the synthesiser to perform a special action such as a slight pause or sound effect.

You can call **SamGetCharSet** to retrieve the set of special characters. The actual Unicode values for each special character is up to the driver providing there is no clash with the other characters sets reported in **SamGetCharSet**, so you must retrieve this list each time you open the device. I.e. the code for a special character can be the same as a valid Unicode character code providing that the Unicode character is not included as one of the alphabetic, modifier or punctuation characters.

When you call **SamGetCharSet** you get a set of records. Each one is a **SPECIALCHAR** structure that is defined in ***sam.h***. These consist of the Unicode value uc and the meaning. The meaning is encoded as a single 32-bit DWORD value.

Sound Effects

Sound effects should have a meaning value of between 0x80000000 and 0x80001000. It is irrelevant what the actual range of sounds can be produced because this is a speech device, not a music synthesiser!

The driver should ideally provide no more than 20 different sound effects.

Musical Notes

If the device is capable of playing musical tones then it will return meaning values of 0x810000aa where aa indicates the note number. There should be a progression of frequencies available where each increase in value represents one semitone. C4 has a value of 60 (3ch).

Therefore, C3 will be 48, E4 will be 64 etc.

Each musical time should be of approx. 200ms duration. To play a longer note multiple codes can be concatenated together in the input string.

Pauses

Pause codes should have a meaning value of between 0x00000000 and 0x00001000 where the meaning value indicates the approximate pause length in milliseconds.

Short pauses are useful for breaking up phrases where a comma may produce too long a pause.

Telephone Tones

If Digital tone multi-frequency (DTMF) telephone tones can be generated by the synthesiser then values from 0x00010000 to 0x0001000f will be specified (there are 16 different possible tones. "0123456789*#ABCD")

2.19 Phoneme support

If the synthesiser supports phonetic characters then it will add the appropriate Unicode characters into its alphabetic character set. Unicode defines a set of standard codes that represent common phonetic symbols. (U+0250 to U+02A8)

There is no special 'phonetic mode'. Any phonetic characters are simply part of the language character set.

2.20 Punctuation and Spelling

The synthesiser will determine how to speak text depending on the character set each character belongs to (as reported in **SamGetCharSet**) and whether spelling is enabled.

Normally the synthesiser will form words from characters in the **alphabetic** set and use the **modifier** letters to adjust the sound of the words. The modifier letters should not produce any speech themselves, regardless of where they appear in the text.

The **punctuation** characters will always be spelt by the synthesiser. If punctuation characters appear in the middle of a word then the synthesiser will break up the word into two.

You can use spell mode to cause the synthesiser to spell out all the characters it receives, providing they are in one of the character sets. **Special** characters are not included in this (they are never 'spelt'). Spell mode is activated by a flag in the call to **SamAppend**.

2.21 Installation

SAM will be installed from the SAM installation disk. The disk will contain SAM and all drivers. SAM will be able to install additional drivers at a later date.

SAM.EXE goes in the SAM directory. Each driver is stored in a subdirectory under the location of SAM.EXE. SAM scans each sub-directory looking for ***.sdi*** files that tell SAM which DLL to use as a driver.

2.22 Management

You should provide the facility in your application to activate SAM's control panel. You can call **SamControl** to do this. This function provides a number of services, including SAM configuration, driver installation, current driver configuration and device selection. If the hardware has changed then SAM will send you notification messages to allow you to check for new synthesisers and Braille displays.

2.23 Sharing Manager

SAM provides a facility for several applications to share the usage of a single synthesiser or Braille display. An interface lets each application co-operatively interact to control the speech and Braille the user hears, whilst still allowing each application complete control over the access device.

The sharing system is designed to be as simple as possible to implement, only a simple message handler is needed. The important thing is that when the access device is being shared, some function calls may fail.

SAM allows for four different types of application;

- A screen reader – An application that attempts to make all other applications, including itself, talk (or Braille). Only one screen reader is allowed to be loaded at once.
- A fully talking application – A program which can provide all necessary speech (or Braille output) for itself, and wants to silence any screen readers that are loaded, whilst it is the foreground application.
- A partially talking application – A program that can produce some speech or Braille, but may want a screen reader to help. Such an application will want to selectively decide if it can speak or the screen reader should speak.
- A background talking application – A program that produces speech (or Braille) on demand by the user, usually by means of a hot key. Such an application could be a background document reader, allowing a user to read the manual whilst using their application. Such applications should not have an active window or have speech or Braille output that relates to screen events.

The sharing system is independent of the synthesiser or Braille display. It switches between applications. If the user has two synthesisers then SAM will prevent them both talking at the same time, even if the two applications have a synthesiser each.

Initialisation

The sharing manager is built into SAM. When you call **SamInit** you must tell SAM what type of application you are, and specify a window for notification messages to be received.

The Screen Reader

A screen reader's job is to make all applications accessible, except those that can do this by themselves. SAM allows only one screen reader to be loaded at a time.

Screen readers typically have a 'screen review' facility. This allows the user to move a review cursor around the screen that is independent of the application. You would probably want to be able to use review mode to review a talking applications screen. Normally SAM will block all screen reader activity whilst a talking application is in the foreground. To make screen review mode work SAM has a facility to allow a screen reader to override the sharing system and speak whilst a talking application is in the foreground. This should not be a problem, as normally the talking application will not be saying anything because nothing on the screen is changing and it has no input. When the screen reader says something in override mode, the application will be notified that it has lost its speech in the normal way so it can stop document reading if it is doing so.

To override the sharing system the screen reader should call **SamControl** specifying the override on and off flags.

Talking Applications

A talking application provides its own speech (or Braille). SAM only allows an application access to the synthesiser or Braille display if it is the foreground application (i.e. the active window). If the user switches away from a talking application SAM will remove its access device control.

A fully talking application should produce some speech or Braille output for everything that happens. Whilst it is the foreground window, SAM will block output from any screen reader that is loaded.

A partially talking application is one that provides speech or Braille output some of the time, but wants priority over a screen reader when it does. When something happens on the screen both the application and a screen reader may try to speak (or Braille). SAM will automatically switch from one to the other. If the application says nothing SAM will allow the screen reader control. If the application has something to say then SAM will block the screen reader.

A background talking application is allowed by SAM to talk or Braille at any time, but may be immediately overridden by any other SAM output.

The Alt-Tab box is treated by SAM as belonging to the screen reader, even though an application keeps the focus until the box disappears.

Device Switching

On each request to speak (or mute or Braille), SAM determines who has priority. If the 'owner of the speech or Braille' has changed, SAM will first mute all synthesisers and then post a message to the old application. The application that has lost control will receive a **SAM_LOST_SPEECH** message.

The purpose of this message is to inform an application that may be document reading (indexing only), that the speech has stopped. The application can then do the same thing as if the user had stopped the process.

There is no message sent to the application that gains control. It will know this anyway because its speech or Braille output calls will start to succeed.

Any attempt to speak or Braille by an application that is not in control will result in the error code **SAM_CANT_SPEAK**.

Calls to **SamIndex** will continue to work (returning the index value at the point the speech stopped) so having lost control of the synthesiser you can still get the index marker.

Notification Messages

To avoid clashes with windows messages and your own private messages, all notifications are sent with a single message number, and the notification code in the lowest 8 bits of wParam. You must specify the message number that you want to receive when you call **SamInit**.

If you receive a **SAM_RESPOND** message, you must call the **SamRespond** function. This message is required to allow SAM to synchronise speech output and screen events from a screen reader and a partially talking application. It also allows SAM to wait until your application has finished, as the message will be posted to the end of the message queue.

If you don't respond correctly to a SAM_RESPOND message then SAM may be unable to synchronise correctly between applications and a screen reader. In addition, SAM may block all screen reader speech and hang your machine waiting for a response.

Hardware changes

SAM has the facility for the user to make changes to their synthesiser configuration, or even change synthesiser without restarting their application or screen reader.

They may want to change to a device with a different language capability or simply change where their synthesiser is plugged in.

If they do this then your application will receive two messages, a **SAM_CONFIG_START** message when the configuration process starts and a **SAM_CONFIG_END** when it finishes.

When you receive a **SAM_CONFIG_START** message you must close all speech units, and don't try to use SAM. Any functions you call may return **SAM_INVALID_HANDLE** or **SAM_INVALID_UNIT**

When you receive **SAM_CONFIG_END** you should then query the available synthesisers and start up your speech in the same way as you would when your application starts. You need to re-enumerate all parameters because you may now be using a different synthesiser.

Braille displays

If you are using a Braille display, you will receive additional SAM notification messages whenever one or more of the buttons on the Braille display is pushed. The second byte of wParam will contain the unit number generating the button push, and the third byte will contain the strip number..

lParam contains information on the button pressed that can be used to obtain the full details via **SamGetButtonPressed**. The type of information returned will depend on the type of the strip.

Because Braille displays often possess several buttons, there are potentially thousands of button combinations. However, you can call **SamGetButtonCombinationStrip** to obtain sensible combinations for performing certain common operations. See **SamGetButtonCombinationStrip** for a discussion of the meanings of the various defined actions.

Braille display button pushes are send to the client who last wrote something to the braille display.

Software Modification

To use the sharing manager, you MUST implement the **SAM_RESPOND** message in your window procedure. If you perform document reading using indexing then you should respond to a **SAM_LOST_SPEECH** message and do the same thing as if the user had hit the stop key. If you are using a Braille display, you should also handle the various button messages from SAM.

Example.

```
switch(msg)
    {
    case SAM_MSG:
            switch(wParam & 0xff)
                    {
                    case SAM_RESPOND:
                            SamRespond();
3                           break;
                    case SAM_LOST_SPEECH:
                            StopDocumentRead();
                            break;

                    }
            break;
    ...
    }
```

SAM_MSG above is the value you specified for message when you called **SamInit**.

When you call **SamSpeak**, **SamAppend** or **SamMute** then you may get a **SAMERROR_CANT_SPEAK** message. This means that you don't have control over the synthesiser.

2.24 Languages

You need to maintain two lists of languages. One is a list of spoken languages, this is needed for **SamAppend** and **SamGetCharSet**. The other is a list of languages that descriptions are available for. This is obtained with the **SamGetLangId** function and is used with the various **SamQuery** functions.

Be careful that you don't mix up these two language lists.

To find out which languages a synthesiser can speak in, you must:

- Call **SamQuerySynth** to find out the number of parameters.
- Repeatedly call **SamQueryParam** to find which parameter specifies the language.
- Repeatedly call **SamQueryParamValue** for the range of the language parameter to get the actual language id for each spoken language.

To find out which languages descriptions are available in, you must:

- Call **SamQuerySynth** to find out the number of languages.
- Repeatedly call **SamGetLangId** to get each language id.

The meaning of the language id is the same for both lists (defined in ***sam.h***).

3. SAM32 Function Reference

This section documents all the functions exported by SAM32.DLL

3.1 SamInit

This function should be called first, before any other functions.

```
DWORD SamInit(
    DWORD type,
    HWND  hwnd,
    WORD  message,
    );
```

Parameters

type

This specifies what type of application you are. You can be either a screen reader, a fully talking application, a partially talking application or a background talking application.

The values for this are **SAM_SR, SAM_FTAP, SAM_PTAP** and **SAM_BTAP** respectively.

hwnd

This is a window handle to receive messages from SAM. You must create a window to receive notification messages from SAM, so you will need a **GetMessage** or **PeekMessage** loop in your application. You can create an invisible window if you like. If your speech thread is unable to have a window or a message queue (i.e. it normally idles, waiting for a synchronisation object) then you can specify a window handle for any other thread you have, providing you have a method of waking up and sending information to your speech thread.

message

To prevent clashes of private message identifiers, you can specify the message id that you will receive. You should set this value to **WM_USER** + some constant value of your choice. The value will be referred to as **WM_SAM**.

Return Value

If the function succeeds, the return value is zero. Otherwise, one of the following error codes may be returned.

SAMERROR_NOSAM – if SAM could not be located or connected to.

SAMERROR_SRLOADED – if there is already a screen reader loaded.

SAMERROR_ALREADYOPEN – if the current thread has already called **SamInit**. A single thread cannot be both a screen reader and a talking app, or a deadlock situation may arise in SAM. If you want to have a single process that is both then you must use more than one thread.

Remarks

This function locates and starts up SAM if it is not already running.

3.2 SamEnd

Call this function when you have finished using SAM. It will allow SAM to free up and unload drivers etc.

```
void SamEnd(void);
```

Remarks

There are no parameters and no return value.

3.3 SamVersion

This function returns the current version of the SAM application, which can be used to determine the API supported.

```
DWORD SamVersion(void);
```

The version number is returned as a 32bit DWORD with the most significant 8 bits corresponding to the major API version number, and the next most significant 8 corresponding to the minor API version number. The lower 16 bits may be zero or contain build specific information, which should not be interpreted but displayed for reference in about boxes etc.

e.g. SAM version 2.01 is encoded 0x0201xxxxx

3.4 SamRespond

This function should be called in response to a **WM_SAM** message with a **SAM_RESPOND** code.

```
void SamRespond(void);
```

Remarks

There are no parameters and no return value. This function will return immediately, and is needed to provide timing synchronisation between a screen reader and a partially talking application.

3.5 SamControl

This function provides configuration control for SAM and synthesisers.

```
DWORD SamControl(
    DWORD code,
    DWORD unitid
    );
```

Parameters

unitid

The meaning of this depends on the value of code.

code

This can be one of the following values:

SAMCONTROL_SAM - Activates the main SAM control panel. This is asynchronous - the function will return immediately. unitid is ignored.

SAMCONTROL_NUM_SYNTH – SAM will return the number of synthesiser units available. Units are numbered from 1 upwards. unitid is ignored.

SAMCONTROL_NUM_BRAILLE – SAM will return the number of Braille display units available. Units are numbered from 1 upwards. unitid is ignored.

SAMCONTROL_DEVICE_SYNTH – Activates the control dialogue box for a given synthesiser unit, specified in unitid. This is asynchronous - the function will return immediately.

SAMCONTROL_DEVICE_BRAILLE – Activates the control dialogue box for a given Braille display unit, specified in unitid. This is asynchronous - the function will return immediately.

SAMCONTROL_SELECT_SYNTH – Causes SAM to display a synthesiser selection list dialogue box. Use this function to provide synthesiser selection for the user if there is more than one synthesiser available. unitid specifies the default selection, providing it is in the range of 1 to the number of units. Otherwise, the default selection will be unit number 1.

SAMCONTROL_SELECT_BRAILE – Causes SAM to display a braille selection list dialogue box. Use this function to provide braille display selection for the user if there is more than one braille display available. unitid specifies the default selection, providing it is in the range of 1 to the number of units. Otherwise, the default selection will be unit number 1.

If there are no units available then this function will return immediately with the code 0xffffffff and no dialogue box will be displayed.

SAMCONTROL_OVERRIDE_ON. Only a screen reader can use this flag. With override on, a screen reader is allowed to talk at any time, especially whilst a fully talking application is in the foreground. Use this flag to allow the screen reader to speak during screen review mode.

SAMCONTROL_OVERRIDE_OFF - This turns off the screen readers override.

SAMCONTROL_QUIT - SAM will be unloaded from memory if no applications are using it. unitid is ignored. You can call **SamControl** with this code after calling **SamEnd**.

Return value

The return value is zero for **SAMCONTROL_SAM** and **SAMCONTROL_DEVICE_…**.

For **SAMCONTROL_SELECT_…** the return code is the unit number of the selected device, or `0xffffffff` if the user cancelled the selection.

For **SAMCONTROL_NUM_…** the return code is the number of units available. This may be zero!

3.6 SamGetCharSet

This is called to determine the set of characters that can be used to make up speech.

```
DWORD SamGetCharSet(
    DWORD  unitid,     // unit identifier
    DWORD  charset,    // charset number
    DWORD *buffer,     // pointer to buffer
    DWORD *size,       // pointer to DWORD to
                       // receive size
    DWORD  language    // requested language
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive).

charset

The identifier of the requested character set. This should be set to **SAMSET_ALPHABETIC, SAMSET_MODIFIER, SAMSET_PUNCTUATION** or **SAMSET_SPECIAL,** to request one of the four character sets.

buffer

This points to a section of memory where the information will be stored, (or it can be NULL). This memory must be allocated before the function is called. The size of this memory is dependent on charset.

size

This must hold the address of a DWORD to receive the size of the required buffer if buffer is NULL.

language

The requested language for the character set. This can be any valid language identifier. SAM will return a character set for any language that can be spoken. This is not the same as the description languages.

Different languages can have different character sets.

Return value

The return value is zero if the function succeeded or an error code otherwise.

SAMERROR_INVALID_ADDRESS if the address pointed to by buffer or size is not valid or doesn't have write access.

SAMERROR_INVALID_UNIT - the unitid value was invalid.

SAMERROR_INVALID_LANGID - the langid did not match any of the speech languages.

Remarks

If buffer is NULL then the function stores the number of characters in the given character set in the location pointed to by size. This may be zero. You should multiply this number by the size of each record to get the buffer size in bytes.

For alphabetic, modifier and punctuation character sets, each record consists of one Unicode value (stored as a WCHAR). The buffer will therefore contain a simple array of WCHAR's, one for each character.

Special characters are stored as an array of **SPECIALCHAR** structures.

The character set will be in ascending order of Unicode values.

3.7 SamAppend

This is the main function used to send text to the synthesiser driver.

```
DWORD SamAppend(
    HANDLE  hsynth,     // handle to open unit
    WCHAR *text,        // pointer to unicode text
    DWORD  length,      // number of characters
    DWORD  index,       // index identifier
    DWORD *vblock,      // pointer to parameter
                        // block
    DWORD  flags        // speech flags
    );
```

Parameters

hsynth

This is a handle to an open unit.

text

A pointer to the start of the Unicode string which should be output. The string does not need to be zero terminated.

The characters in this string should be contained within the character set of the specified language. Characters not in the character set will be ignored. You can get the character set with the **SamGetCharSet** function.

length

The length is the number of characters in the text string. The size of the buffer pointed to by text must therefore be at least length`*sizeof(WCHAR)`.

To perform indexing the client can subdivide a source string into individual words and call **SamAppend** with each word, without having to allocate or manipulate any additional memory.

There is no limit to the length of the text!

If length is zero, the string is assumed to be NULL terminated. (NULL meaning 2 bytes, both zero).

index

The index value is a unique value to identify this piece of text. If you want to perform indexing then this is the identifier that will be reported back in a call to **SamIndex**.

It is up to an application to ensure that the index identifier is unique for all pieces of text, unless it does not intend to do indexing, in which case index can be zero or some other constant value.

vblock

This is a pointer to a voice block for the speech. All of the text in this call to **SamAppend** will use the speech parameters in vblock. If this parameter is NULL, the driver will use the same parameters as the last call to **SamAppend** (or the default parameters if there has never been a call to **SamAppend**), however this may be different if another application has used the device in the mean time. If you want full control over the voice, you should always provide a voice block.

Multiple calls to **SamAppend** can contain the same address for vblock, however the driver will make a copy of any relevant data so that you can change the contents of *vblock.

flags

The following flags are defined, which give additional information to the driver about how text should be processed.

SYNDTEXT_RAW – if this is set then the driver will send the text directly to the synthesiser without any pre-processing. This may be used by the client to send special synthesiser dependent commands. With the right screen reader software users will be able to embed synthesiser commands in their documents. Not all devices are capable of implementing this. You should check in the **synthparams.caps** for the presence of the **SAMCAPS_RAWOUTPUT** flag. This flag overrides the spell flag.

SYNTEXT_SUPPRESS – if this set the driver will not speak the associated text, but will process it for context information. This allows words to be spoken in the context of a larger grammatical structure, by passing in the preceding and following text with this flag set, and the words of interest with it clear. The client should check for the presence of the **SAMCAPS_SPEAKINCONTEXT** flag in **synthparams.caps** before using this feature, although SAM will ensure text with the flag set is not spoken by synthesisers that do not support the feature. Note that indexes will not be returned for any text with this flag set.

SYNDTEXT_SPELL – if this is set then the synthesiser will spell all characters in this section of text. This includes the alphabetic, modifier and punctuation characters. Special characters are not included. The characters received will still be restricted to the character set reported by the driver in **SamGetCharSet**. The pronunciation of letters is the standard for the language in use; for example, in English “cat” would be “see, ay, tee”.

SYNTEXT_SPELLPHONETIC and **SYNTEXT_SPELLATC.** Either of these flags can be set in conjunction with the **SYNDTEXT_SPELL** flag to alter the pronunciation of letters. The client should check for the **SAMCAPS_SPELLPHONETIC** and **SAMCAPS_SPELLATC** flags respectively, to determine if the driver supports this feature. **SYNTEXT_SPELLPHONETIC** uses the letter sounds, for example in English “cat” would be “cuh, ah, tuh”. **SYNTEXT_SPELLATC** uses the international Air Traffic Control names for the letters, e.g. “cat” would be “charlie, alpha, tango”.

SAMCAPS_RAWOUTPUT – if this is set it signifies that the synthesiser is able to recognise currency values in the text, and to speak them in the appropriate manner for the current language.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_HANDLE – hsynth is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by text is not valid or doesn't have read access.

SAMERROR_CANT_SPEAK – if this is returned then your application has not produced any speech because SAM's sharing manager has determined that another application has control of the synthesiser. Your text will not have been added to the buffer, and the buffer will have been emptied. The synthesiser will also have been muted and is now under the control of another application. You can however, call **SamIndex** to find out how far your last section of speech got before it was muted.

Remarks

This function will only queue the speech output and not actually produce any speech. You must call **SamSpeak** before any speech can be generated.

The text string pointed to by text should only contain characters specified in the relevant character sets reported by the driver. Any characters not in these sets will be ignored by the driver.

All text strings in multiple calls to **SamAppend** will be treated as a single phrase and concatenated together. The only reason you might want to break up a sentence into multiple calls to **SamAppend** is to perform indexing or to change a parameter in mid-sentence.

The client may call **SamAppend** while speech is in progress (i.e. the synthesiser is talking). In this case, the driver will queue the text ready to append it onto the end of the current section of speech. If **SamSpeak** is called after more text has been queued, then the driver will append it onto the current speech. The index value in the original call to **SamSpeak** will be discarded and the end of speech index value will be taken from the latest call to **SamSpeak**.

3.8 SamMute

This is used to stop any speech in progress and clear any pending speech.

```
DWORD SamMute(
    HANDLE hsynth
    );
```

Parameters

hsynth

This is a handle to an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_HANDLE – hsynth is not valid.

SAMERROR_CANT_SPEAK – This is returned if SAM's sharing manager has determined that another application has control of the synthesiser.

Remarks

This should halt speech immediately. You can then find out the index position that the speech reached with a call to **SamIndex**.

This function may also be called after a number of calls to **SamAppend** without a following **SamSpeak.** In this case, the synthesiser will stop speaking (if it was) and the speech queue will be cleared.

3.9 SamSpeak

This function is called to start the speech of text from one or more previous calls to **SamAppend**.

```
DWORD SamSpeak(
    HANDLE hsynth,
    DWORD  index,
    DWORD  flags
    );
```

Parameters

hsynth

This is a handle to an open unit.

index

The index value is a unique value that is used to identify the end of this speech block. This is the value that will be returned in a call to **SamIndex**. You can specify any DWORD value you like in here.

flags

There are no flags defined so this parameter must be zero.

Return Value

If the function is successful, the return value is zero. Otherwise the return value may be one of the following.

SAMERROR_INVALID_HANDLE – hsynth is not valid.

SAMERROR_CANT_SPEAK – if this is returned then your application has not produced any speech because SAM's sharing manager has determined that another application has control of the synthesiser. Your speech buffer will have been emptied. Calls to **SamIndex** will continue to return the last index value from the last time you had control of the synthesiser and will be the index value at the point that speech finished, or was muted by SAM's sharing manager.

This function also implies end of phrase.

3.10 SamIndex

This function retrieves the index identifier for the currently speaking block of text and other information.

```
DWORD SamIndex(
    HANDLE hsynth,
    DWORD *index,    // address of DWORD to
                     // place index
    DWORD *flags     // address of DWORD for
                     // flags
    );
```

Parameters

index

This is a pointer to a DWORD to receive the index identifier. This is the value that was passed in the call to **SamAppend** that the currently speaking text came from. If the speech has finished then the value retrieved will contain the value specified in the call to **SamSpeak.** If the speech has not yet started (its a really slow device...) then it will return the index value of the first word.

If the synthesiser is unable to do word or phrase indexing then it should store 0 at this location. Indexes will not be returned for text passed to **SamAppend** to with the **SYNTEXT_SUPPRESS** flag set.

flags

This driver will set the following flags as appropriate.

SYNDSPEECH_INPROGRESS – should be set if the synthesiser is speaking, or has more to say.

Return Value

The return value should be zero if successful.

Remarks

You should check the **synthparams.caps** structure to check what at resolution the speech progress information is available.

If your application lost control of the synthesiser then you can still call this function to find out what was last spoken.

3.11 SamOpenSynth

SamOpenSynth is used to open a handle to the speech unit specified. Once the unit has been opened then the handle value returned is used to identify the device.

```
DWORD SamOpenSynth (
    DWORD   unitid, // unit number
    HANDLE *handle  // synth handle
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive).

handle

This is a pointer to a HANDLE to receive the handle of the new unit (if it was opened successfully).

Return Value

The function returns zero if it succeeds.

If the unit failed to open the return value is one of the following.

SAMERROR_INVALID_UNIT – the unitid value was invalid.

SAMERROR_UNIT_FAIL – the physical device is not responding, turned off, unplugged or otherwise unusable.

Remarks

The handle returned is the handle of the unit if successfully opened.

The handle should be used in all subsequent SAM calls. The handle may be any arbitrary value.

You should call **SamCloseSynth** to release ownership of the device.

3.12 SamCloseSynth

SamCloseSynth is called to close access to the unit. You should call this function when you have finished with the device (usually when your application has terminated).

```
DWORD SamCloseSynth (
     HANDLE hsynth  // synth handle
     );
```

Parameters

hsynth

This is the handle of the unit that was created with **SamOpenSynth**.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_HANDLE – hsynth is not valid.

Remarks

Once the unit is closed, the handle will no longer be valid. If the unit is opened again (with **SamOpenSynth**) it is likely that a different handle will be returned.

If the unit is battery driven or powered from the PC, the driver should (if possible) power-down the unit in this function (to conserve power on a portable system).

3.13 SamQuerySynth

This function gives additional information about each unit. You can call **SamControl(SAMCONTROL_NUM_SYNTH)** to determine how many units there are. Unit information can be requested at any time without first opening the unit or while the synthesiser is talking.

```
DWORD SamQuerySynth (
    DWORD         unitid, // synth unit number
    SYNTHPARAMS *lpunit, // address of
                         // SYNTHPARAMS sruct
    DWORD         langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range of 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

langid

This specifies the language that the synthesiser description string is reported in. English (44) must be available.

Return Value

This function fills the parameter block pointed to by lpunit with the units parameters.

The function returns zero if it succeeded.

If unitid is not valid (regardless of the validity of lpunit), the return value is **SAMERROR_INVALID_UNIT**.

SAMERROR_INVALID_ADDRESS will be returned if the address pointed to by lpunit is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID will be returned if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

SAMERROR_NOT_A_SYNTH will be returned if the unit is not a synthesiser.

Remarks

See **SYNTHPARAMS** for a description of the parameters structure

3.14 SamQueryParam

Each synthesiser unit may have a variable number of parameters. The parameters for one synthesiser may not match any other synthesiser. **SamQueryParam** allows you to enumerate all the available parameters with descriptions. This interface allows additional parameters to be added in the future, and be compatible with older software.

It is the driver's responsibility to report correct information about the unit. It can do this by either querying the unit directly or maintaining a data table of parameter information in memory.

SamQueryParam may be called at any time.

```
DWORD SamQueryParam (
    DWORD     unitid,    // unit number
    DWORD     pnum,      // parameter number
    SAMPARAM *param      // address of
                         // parameter block
    );
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

param

Address of a **SAMPARAM** structure to receive the data.

Return Value

The return value is zero if the function succeeded. Otherwise the possible error codes are:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_ADDRESS – if the address pointed to by param is not valid or doesn't have write access.

Remarks

For each parameter the client wants to query, it will allocate the necessary memory for a **SAMPARAM** structure first.

See **SAMPARAM** for a description of a single parameter.

Client code example.

```
...
SAMPARAM param;
for (num=0;num<unitp.params;num++)
    {
    SamQueryParam(unitid,num,&param);
    ...
    }
...
```

3.15 SamQueryParamDesc

This function returns a description for a given parameter.

```
DWORD SamQueryParamDesc (
    DWORD  unitid,        // unit id
    DWORD  pnum,          // parameter number
    DWORD  language,      // requested langid
    WCHAR *description,   // address of output
                          // buffer
    DWORD *size           // size of required
                          // buffer
    );
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

language

The requested language of the description. This should be a value returned from **SamGetLangId.**

description

This is a pointer to a buffer to receive the string. The string is in Unicode.

This value can be NULL if you want to know the length of the description string.

size

This is a pointer to a DWORD to receive the size of the required buffer in WCHARS.

Return Value

If description is NULL then the number of WORDS in the resulting string (including the trailing zero) will be put in size. You can use this value to allocate the necessary memory. If description is not NULL the requested string will be copied to the memory pointed to by description.

If the function succeeds the return code will be zero. Otherwise the function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by description and size will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_ADDRESS – if the address pointed to by description is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if the language requested is not supported.

SAMERROR_NO_DESCRIPTION – if the driver does not have a description for this parameter in the specified language. This may occur if the driver has a partially complete set of descriptions for a particular language.

Remarks

No formatting will be provided. The description will be just the single name of the item, e.g. "Volume", "Voice" or "Speed". The string copied must be in Unicode.

The first letter will be capitalised.

You may use this string to describe the parameter to the user and to identify the parameter when loading and saving voice blocks.

The driver is guaranteed to support a full set of English descriptions. Any other languages are dependent on the driver.

3.16 SamQueryParamValue

This returns the numeric value for **SAMPARAM_COMPOUND** parameters. See the **SAMPARAM** structure for a description of this parameter type.

```
DWORD SamQueryParamValue (
    DWORD  unitid,     // unit id
    long   pnum,       // parameter number
    DWORD  val,        // value index
    DWORD *pvalue      // pointer to DWORD to
                       // receive result
);
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

val

This specifies the parameter value requested. val should be set to an integer from zero to *range*-1 where *range* is obtained from the **SAMPARAM** structure for this parameter.

pvalue

This is a pointer to a **DWORD**. If the function is successful, the result is placed here.

Return Value

If the function succeeded, the return value is zero. Otherwise, one of the following error codes may be returned.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number pnum is out of range.

SAMERROR_INVALID_ADDRESS – if the address pointed to by pvalue is not valid or doesn't have write access.

SAMERROR_INVALID_VAL – if val is beyond the range of valid values.

Remarks

This function is only relevant for parameters that are marked as **SAMPARAM_COMPOUND**. For other parameter types pvalue is the same as val. This function may be called with any valid pnum so for non-**SAMPARAM_COMPOUND** parameters the driver will return val.

3.17 SamQueryParamChoice

This generates a string for a given parameter and value. You can call this function to get a description of each value for a given parameter. This is essential for Multi-choice parameters.

```
DWORD SamQueryParamChoice (
    DWORD  unitid,         // unit id
    Long   pnum,           // parameter number
    DWORD  val,            // actual parameter
                           // value
    DWORD  language,       // requested language
    WCHAR *description,    // address of buffer
    DWORD *size
    );
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

If this is zero or greater then it indicates the requested parameter number that the client wants a description of.

If it is equal to -1 then this function returns the voice name for a given preset voice. The number of preset voices is specified by the voices parameter in the **SynthParams** structure.

val

This is the actual value of the parameter. For numeric and multi-choice parameters, it must be in the range zero to *range*.

For compound parameters, it must be a valid value that has been retrieved with **SamQueryParamValue**.

language

The requested language of the description. This should be a value returned from **SamGetLangId**.

description

This is a pointer to a buffer to receive the string. The string must be stored in Unicode.

If this value is NULL, the client is requesting the length of the description string.

size

This is a pointer to a DWORD to receive the size of the required buffer in WORDS.

Return Value

If description is NULL then the number of WORDS in the resulting string (including the trailing zero) will be put in size. You should use this value to allocate the necessary memory. If description is not NULL the requested string should be copied to the memory pointed to by description.

If the function succeeds, the return code will be zero. Otherwise, the function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by *description* and *size* will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_VAL – if the val parameter is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by description is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if the language requested is not supported.

Remarks

No formatting will be provided. The description is just the single name of the item, e.g. "Male", "Female" or "Off". The string will be stored in Unicode.

Descriptions of numeric parameters may not always contain numbers. For example, volume zero may be described as "Silent" instead of "0".

3.18 SamGetLangId

This is used by the client to enumerate the languages available for descriptions of parameters. This is nothing to do with which languages the synthesiser can speak. The total number of available languages is stored in the **SYNTHPARAMS** structure. The client will call this function with language index number and retrieve a language identifier. The identifier should be used in subsequent calls to **SamQuery...** functions.

```
DWORD SamGetLangId (
    DWORD  unitid,   // unit id
    DWORD  index,    // n'th language
    DWORD *langid
    );
```

Parameters

unitid

The unit id of the synthesiser device.

index

The language index number. This should be in the range from zero to *langs*-1, where langs is from the **SYNTHPARAMS** structure.

langid

This is a pointer to a DWORD to receive the language id code.

Return Value

If the return value is zero then the DWORD pointed to by langid will be filled with the actual language id for that language. The actual meaning of each language id is defined in ***sam.h***. These codes will be common between all drivers so that an application can identify each language.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

If the function succeeds, the return value is zero.

Remarks

The code returned for each supported language will be defined in ***sam.h***.

3.19 SamGetDefaultVoice

This function is used by the client to retrieve the default voice and one or more preset voices.

```
DWORD SamGetDefaultVoice (
    DWORD  unitid,        // unit id
    long   langid,        // spoken language id
    DWORD *voiceblock     // address of voice
    );                    // block buffer
```

Parameters

unitid

The unit id of the synthesiser.

langid

This value specifies spoken language required for the default voice.

voiceblock

This is a pointer to a memory buffer where the driver will store a copy of the voice block for the requested voice.

The size of the memory buffer required is
`sizeof(DWORD)*synthparams.params`

You should allocate the required memory before calling this function.

Return Value

If the function succeeds, the return value will be zero.

The function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by voiceblock will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by voiceblock is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if no preset voices correspond to the langid.

3.20 SamGetVoice

This function is used by the client to retrieve the default voice and one or more preset voices.

```
DWORD SamGetVoice (
    DWORD  unitid,         // unit id
    long   vnum,           // voice number
    DWORD *voiceblock      // address of voice
    );                     // block buffer
```

Parameters

unitid

The unit id of the synthesiser.

vnum

This value specifies the preset voice number you want to get. A value of zero specifies the default voice.

voiceblock

This is a pointer to a memory buffer where the driver will store a copy of the voice block for the requested voice.

The size of the memory buffer required is
`sizeof(DWORD)*synthparams.params`

You should allocate the required memory before calling this function.

Return Value

If the function succeeds, the return value will be zero.

The function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by voiceblock will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by voiceblock is not valid or doesn't have write access.

SAMERROR_INVALID_VNUM – if vnum is not in the valid range for a preset voice number.

Remarks

It is recommended that you enumerate all the voices and cache the voice blocks internally.

Use **SamQueryParamChoice** to get a name for each preset voice.

3.21 SamTranslateBraille

A client can use this function to translate a unicode string into the device independent Braille representation used in the other SAM Braille functions.

```
DWORD SamTranslateBraille(
    WCHAR *text,        // Unicode text
    DWORD  len,         // Length
    WORD   *braille,    // Pointer to hold Braille
                        // equivalent
    WORD    unknown     // Dot pattern to use for
                        // unknown characters
    );
```

Parameters

text

A pointer to a unicode string to translate into Braille.

len

The length of the string to translate.

braille

A pointer to store the Braille translation of text, this should be len WORDs long.

unknown

The dot pattern to substitute for characters that have no Braille translation under this function.

Return Value

This function will return 0 on success or one of the following error codes:

SAMERROR_INVALID_ADDRESS – the memory pointed to by text or braille is invalid, or text does not have read access or braille does not have write access.

Remarks

The translation is a one-to-one mapping of text characters to Braille dot patterns. This function will only successfully translate those characters in the range U+0020 - U+007f. Any other characters will be translated as the unknown character.

The format of the unknown character is the same for characters passed to **SamSetDisplayStrip**.

The Braille character set used is the North American Braille Computer Code. If other language tables are required, the application will have to do its own translation.

3.22 SamQueryBrailleEx

This function gives information about each Braille unit. You can call **SamControl(SAMCONTROL_NUM_BRAILLE)** to determine how many units there are. Unit information can be requested at any time without first opening the unit or while the Braille display is in use.

```
DWORD SamQueryBrailleEx(
    DWORD             unitid,
    BRAILLEPARAMSEX *lpunit,
    DWORD             size,
    DWORD             langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

size

The client should set this to the size in bytes of the currently defined **BRAILLEPARAMSEX** structure. The structure may be extended in future versions of SAM, but this field allows SAM to only return as much data as the client is aware of.

langid

This specifies the language that the Braille display description string is reported in U.K. English (44) is guaranteed to be available.

Return Value

This function fills the parameter block pointed to by lpunit with the units parameters. The function returns zero if it succeeded, one of the following error codes:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpunit).

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpunit is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

SAMERROR_WRONGSAMVERSION – if the size is larger than SAM expects, this indicates that the client needs a newer version of SAM to operate correctly.

Remarks

See **BRAILLEPARAMSEX** for a description of the parameters structure.

The driver should check that all parameters are valid and return the appropriate error code if they are not.

3.23 SamQueryStrip

This function gives information about each strip supported by the Braille unit. The number of strips is contained in **brailleparamsex.strips** returned by **SamQueryBrailleEx**. Strip information can be requested at any time without first opening the unit or while the Braille display is in use.

```
DWORD SamQueryStrip(
    DWORD        unitid,
    DWORD        strip,
    STRIPPARAMS *lpstrip,
    DWORD        size,
    DWORD        langid);
```

Parameters

unitid

This is the unit id of the device to query in the range 1 to the number of units.

strip

This is the index of the strip to query in the range 0 to the number of strips-1.

lpstrip

This is a pointer to the parameter block to be filled or NULL.

size

The client should set this to the size in bytes of the currently defined **STRIPPARAMS** structure. The structure may be extended in future versions of SAM, but this field allows SAM to only return as much data as the client is aware of.

langid

This specifies the language that the strip description string is reported in. U.K. English (44) is guaranteed to be available.

Return Value

This function fills the parameter block pointed to by lpstrip with the strips parameters.

The function returns zero if it succeeded, one of the following error codes:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpstrip).

SAMERROR_INVALID_STRIP – if strip is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpstrip is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

SAMERROR_WRONGSAMVERSION – if the size is larger than SAM expects, this indicates that the client needs a newer version of SAM to operate correctly.

Remarks

See **STRIPPARAMS** for a description of the parameters structure.

3.24 SamGetBrailleLangId

This is used by the client to enumerate the languages available for descriptions of parameters. The total number of available languages is stored in the **BRAILLEPARAMSEX** structure. The client will call this function with language index number and retrieve a language identifier. The identifier should be used in subsequent calls to **SamQuery...** functions.

```
DWORD SamGetBrailleLangId(
    DWORD  unitid,     // unit id
    DWORD  index,      // n'th language
    DWORD *langid
    );
```

Parameters

unitid

The unit id of the device.

index

The language index number, this should be in the range from zero to langs-1, where langs is from the **BRAILLEPARAMSEX** structure.

langid

This is a pointer to a DWORD to receive the language id code.

Return Value

If the return value is zero then the DWORD pointed to by langid will be filled with the actual language id for that language. The meaning of each language id is defined in ***sam.h***. These codes should be common between all drivers so that an application can identify each language.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

If the function succeeds, the return value is zero.

Remarks

The driver must support English. The code returned for each supported language should be defined in ***sam.h***.

3.25 SamOpenBraille

SamOpenBraille is used to open a Braille display unit. Only one process can open a unit at a time.

```
DWORD SamOpenBraille(
    DWORD   unitid,    // unit number
    HANDLE *handle     // Display handle
    );
```

Parameters

unitid

The unit number from one to the number of units (inclusive).

Handle

Returned handle to use for subsequent calls to **SamSetDisplayStrip**, **SamSetCursorStrip** and **SamClearDisplayStrip**.

Return Value

The function returns zero if it succeeded.

If the unit failed to open the return value is one of the following:

SAMERROR_INVALID_UNIT – the unitid value was invalid.

SAMERROR_UNIT_OPEN – the unit is already open.

SAMERROR_UNIT_FAIL – the physical device is not responding, turned off, unplugged or otherwise unusable.

Remarks

You should call **SamCloseBraille** to release ownership of the device.

Only one client is allowed to open each unit at one time.

3.26 SamCloseBraille

Used to close the unit.

```
DWORD SamCloseBraille(
    HANDLE hBraille
    );
```

Parameters

hBraille

The handle of an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_UNIT_CLOSED – the specified unit was not open.

Remarks

Once the unit is closed, the handle will no longer be valid. If the unit is opened again (with **SamOpenBraille**) it is likely that a different handle will be returned.

If the unit is battery driven or powered from the PC, the driver should (if possible) power-down the unit in this function (to conserve power on a portable system).

3.27 SamSetDisplayStrip

This is the main function used to send text to a Braille display strip.

```
DWORD SamSetDisplayStrip(
    HANDLE hBraille,
    DWORD  strip,
    WORD   *dispdata
    );
```

Parameters

hBraille

The handle to an open unit.

strip

The strip number.

dispdata

This points to a block of words representing the dot pattern to be displayed on the cells of the strip. Dispdata should contain the same number of WORDs as Braille cells as reported by **SamQueryStrip**.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

SAMERROR_INVALID_ADDRESS – the address pointed to by dispdata is not valid or doesn't have read access.

Remarks

Each strip that a unit returns may have it display set independently using this function.

The format of each WORD in dispdata in a standard form as follows:

bit 0 - dot 1	bit 8 - blink dot 1
bit 1 - dot 4	bit 9 - blink dot 4
bit 2 - dot 2	bit 10 - blink dot 2
bit 3 - dot 5	bit 11 - blink dot 5
bit 4 - dot 3	bit 12 - blink dot 3
bit 5 - dot 6	bit 13 - blink dot 6
bit 6 - dot 7	bit 14 - blink dot 7
bit 7 - dot 8	bit 15 - blink dot 8

This format makes it possible to manipulate cells easily. The dots are arranged in the standard way:

1	●	●	4
2	●	●	5
3	●	●	6
7	●	●	8

If a dot is set to blink, the dot’s normal state (i.e. bits 0-7 above) is periodically exclusive OR’d with the blink state (bits 8-15). This means that it will be possible to have blinking and inverse blinking dots. The distinction may not be clear to a Braille reader.

The data string must be of a fixed length as reported by calling **SamQueryStrip**. The first word represents the leftmost cell. For multiple linked rows of cells, a call to should be made to **SamSetDisplayStrip** for each strip, so they are updates simultaneously from the point of view of the user.

3.28 SamClearDisplayStrip

This is used to clear one or all Braille display strips.

```
DWORD SamClearDisplayStrip(
    HANDLE hBraille,
    DWORD  strip
    );
```

Parameters

hBraille

The handle to an open unit.

strip

The strip number. If a value of **SAMSTRIP_ALL** is used, all strips will be cleared.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

Remarks

All cells on the Braille affected braille strips will be blanked, and any cursor(s) will also be removed.

3.29 SamSetCursorStrip

This is used to position, set the appearance and blink rate of a cursor.

```
DWORD SamSetCursorStrip(
    HANDLE hBraille,
    DWORD  strip,
    long   pos,
    WORD   shape,
    DWORD  rate
    );
```

Parameters

hBraille

The handle to an open unit.

strip

The strip number or **SAMSTRIP_ALL**.

Pos

The position of the cursor in the range 0 to the number of cells - 1, 0 is the leftmost cell in the display area. A value of **SAMCURSOR_HIDE** turns off the cursor on the particular strip, and if strip is **SAMSTRIP_ALL** all cursors on all strips are turned off.

Shape

The dot pattern of the cursor. When the cursor is "on", this pattern is OR'd with the dot pattern of the cell under the cursor.

Rate

The blink rate of the cursor from 1 to the maxrate parameter as reported by **SamQueryBrailleEx**. A value of 0 yields a static cursor. 1 is the slowest blink rate.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBrailleis not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

SAMERROR_INVALID_VAL – pos or rate is out of range.

Remarks

The cursor will be repositioned and the display data unaffected.

The blink rate of the cursor will also affect any blinking text on the display.

See **SamSetDisplayStrip** for the format of shape.

3.30 SamQueryButtonStrip @@@

This function is used to get a description of each general-purpose key, or the possible combinations of routing button presses associated with each cell.

```
DWORD SamQueryButtonStrip(
    HANDLE unitid,
    DWORD  strip,
    DWORD  num,
    WCHAR *description,
    DWORD *size,
    DWORD  langid
    );
```

Parameters

unitid

The number of a Braille display unit.

strip

The strip number.

Num

For strips of type **SAMSTRIP_KEYS** this is the button index from 0 to the number of buttons-1, to get a description for. For all other strip types the call will return descriptions for the possible button combinations. num should range from zero to the number of button combinations-1 in a strip. The button combinations are returned by **SamQueryStrip**.

Description

The address where to store the description, or NULL.

Size

The address to store the length of the description. If this value is zero on exit there is no description available. langid

The language code of the description.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_STRIP – if the strip is not valid.

SAMERROR_INVALID_ADDRESS – if the address of description is invalid or does not have write access.

SAMERROR_INVALID_LANGID – if the language id is not valid.

Remarks

If description is NULL, this function returns the size of the description only.

This function can be used regardless of whether the unit is open or in use by another application.

For strips of type **SAMSTRIP_KEYS** each key may be given an individual name by the driver, and returned to the client by this function. For the other strip types, the client should name the keys using the strip type and the button index, and then append the descriptions of the possible button combinations returned by this function.

For example, the descriptions of button combination values might be:-

0. "Primary routing button"
1. "Auxiliary routing button"
2. "Primary routing button with shift"
3. "Auxiliary routing button with shift"

Drivers may not supply descriptions for all button combinations with strips of type **SAMSTRIP_BUTTONS,** for example when there are a large number of combinations for some of the subtypes such as dials / wheels / 2D. In which case the application should generate a suitable name based on the type and the number.

3.31 SamIsKeyValidStrip

This function determines if a particular combination of general-purpose buttons is valid for the unit.

```
BOOL SamIsKeyValidStrip(
    HANDLE unitid,
    DWORD  strip,
    BYTE   *pkey
    );
```

Parameters

unitid

The number of a Braille display unit.

strip

The strip number. This call if only valid for strips of type **SAMSTRIP_KEYS,** all other types will cause the function to return FALSE.

pKey

A pointer to a byte array forming a bit mask, indicating which buttons are pressed. Bit 0 corresponds to button zero. The size of the array should correspond to the number of buttons on the strip as returned by **SamQueryStrip**.

Return Values

This function returns TRUE if the unitid and strip are valid, and the button combination is valid.

Remarks

There is only one class of buttons that may be pressed in combination (except the action of shifting keys on routing buttons) and these will always be located on a strip of type **SAMSTRIP_KEYS**, however there may be certain combinations of buttons which are not valid (for example a combination of thumb keys and other buttons).

3.32 SamGetButtonCombinationStrip

MISSCHIEN DOOR USER TE KOPPELEN??

This function is used to obtain sensible default button combinations for a variety of client software Braille related features.

```
DWORD SamGetButtonCombinationStrip(
    DWORD  unitid,
    DWORD  action,
    DWORD *strip,
    BYTE  *pbutton,
    DWORD *size
    );
```

Parameters

unitid

The unit number of a Braille unit.

action

One of the actions defined in ***brldacts.h***, see below.

strip

This points to a DWORD to store the returned strip number for the button combination.

pbutton

This points to buffer of size bytes to hold the recommended default button combination. If strip is of type **SAMSTRIP_KEYS** this will be byte array forming a bit mask, indicating which button(s) are pressed. For other strip types it will be a **BRAILLEBUTTONPRESS** structure.

size

A pointer to size of the buffer at pbutton, updated with the size required on exit.

Return Value

This function returns zero on success or one of the following error codes:

SAMERROR_INVALID_ADDRESS – the address pointed to by pbutton is not valid or does not have write access.

SAMERROR_INVALID_VAL – there is no recommended default button combination.

SAMERROR_INVALID_UNIT – the unit number is not valid.

Remarks

The actions defined in ***brldacts.h*** and there associated intended meanings are listed below.

Note that it is likely that a particular display may not support all action codes (because, for example, there aren't enough physical keys or sensible key combinations). If the result is **SAMERROR_INVALID_VAL**, you should ignore the combination.

If pbutton is NULL, only the required size of the buffer will be return in size.

The driver may decide to assign these actions to either a single key or a combination of keys held down for strips of **SAMSTRIP_KEYS,** or it may assign it to an individual button on the other strip types. Therefore, the client must be able to handle this call returning either a bit mask array or a structure.

SAMBRLACT_NO_ACTION	No action. This action does nothing.
SAMBRLACT_WINDOW_TOPLEFT	Move display to top left of window area. This will show the first portion of the first line of information in the current window.
SAMBRLACT_WINDOW_BTMLEFT	Move display to bottom left of window area. This will move the display to the first portion of the last line of information in the current window.
SAMBRLACT_WINDOW_FOCUS	Move display to show the focus. This will move the display to a place such that the current focus (whether it is a highlight, caret or some other visible sign) is somewhere in the Braille.
SAMBRLACT_WINDOW_LEFT	Move display to left edge of window. This will move the display to the left edge of the current line in the window.
SAMBRLACT_WINDOW_RIGHT	Move display to right edge of window. This will move the display to the right edge of the current line in the window.
SAMBRLACT_LINE_UP	Scroll display one line up. The horizontal position, if possible, should not be affected.
SAMBRLACT_LINE_DOWN	Scroll display one line down. The horizontal position, if possible, should not be affected.
SAMBRLACT_CHAR_LEFT	Scroll display one character left. This will stop at the left edge of the line.
SAMBRLACT_CHAR_RIGHT	Scroll display one character right. This will stop at the right edge of the line.
SAMBRLACT_WIDTH_LEFT	Scroll display one window width left. Similar to CHARLEFT, this scrolls by the width of the Braille display. The action stops at the start of the line.
SAMBRLACT_WIDTH_RIGHT	Scroll display one window width right. Similarly, to CHARRIGHT, this scrolls right by the width of the Braille display. The action stops at the end of the line.

SAMBRLACT_HWIDTH_LEFT	Scroll display half a window width left.
SAMBRLACT_HWIDTH_RIGHT	Scroll display half a window width right.
SAMBRLACT_PREVIOUS	Move display to previous data. This scrolls the display one display width to the left if the display is not at the start of the line. If it is at the start of the line it should show the end of the previous line. This is one of the most important actions to implement.
SAMBRLACT_NEXT	Move display to next data. If the display is not at the end of the line, it should scroll by one display width to the right. Otherwise it should display the first portion of the next line. This is one of the most important actions to implement.
SAMBRLACT_TOGGLE_TRACK	Toggle focus tracking. When focus tracking is on, whenever the focus moves outside the display area, the display must move to show the focus. Some users may find it useful to turn off this feature in order to monitor a particular region of the screen.
SAMBRLACT_ATTRIB1	Switch to show first character attributes. This switches the display from showing characters to showing the attributes of the characters. It is up to the client application as to what attribute information to show, but it could include whether the characters are bold or italicised, or the colour of the characters. If this command is issued when the display is already showing attributes, the display should switch back to showing characters.
SAMBRLACT_ATTRIB2	Switch to show secondary character attributes. This functions in a similar manner as ATTRIB1.
SAMBRLACT_REPORT	Switch to show status report. Similarly to the Attribute commands, this command should switch the display to give a brief status report. It is up to the client application as to what status information to present, but it could include, for example, the type of the current focus, the position of a caret, what features of the Braille software are enabled, etc. If this command is issued when the status report is shown, the display should switch back to show characters.
SAMBRLACT_TOGGLE_EIGHTDOT	Switch between six dot and eight dot Braille. Normally, characters are represented in a Braille display using eight dot Braille.

	Some users may find this eight dot Braille difficult to read. Switching to six dot Braille removes the lower two dots (dots 7 and 8) from each character in the display.
SAMBRLACT_TOGGLE_CTYPE	Toggle through available cursor styles. There may be a choice of cursor shapes and blink rates. This command will toggle through the available settings.
SAMBRLACT_TOGGLE_CVIS	Toggle showing the cursor (caret) position. In some circumstances, it may be beneficial to hide the position of the caret. This command toggles this feature.
SAMBRLACT_TOGGLE_AUDIO	Toggle audio feedback. There may be audio feedback coupled with certain Braille actions, such as moving between lines on the display. This command toggles this feature.
SAMBRLACT_TOGGLE_BLANKS	Toggle the skipping of blank lines when navigating. If blank lines are skipped, the PREVIOUS, NEXT, LINEUP and LINEDOWN operations should not land on a completely blank display.
SAMBRLACT_TOGGLE_BRAILLE	Toggle use of Braille display. This effectively turns the use of the Braille display on and off.
SAMBRLACT_TOGGLE_LAYOUT	This toggles between simply showing the characters on the currently displayed line and showing the characters in such a way as to give an indication of their relative position across the screen. With layout mode on, the Braille display is taken to be the width of the window on the screen, however wide that may be.
SAMBRLACT_TOGGLE_TREMBLE	Some users prefer capitalized letters to blink (like the cursor) rather than using eight dot Braille. This action toggles this feature.
SAMBRLACT_ENHANCE	When on, the Show Enhancement feature indicates non-ordinary character attributes by adding dot 8 to each character. This action toggles this feature.
SAMBRLACT_LEFTCLICK	The cursor routing keys on a Braille display can be used to simulate the action of clicking the mouse. This action code indicates that single left mouse clicks should be simulated.
SAMBRLACT_DOUBLECLICK	This action indicates that double clicks should be simulated when the cursor routing keys are pressed.
SAMBRLACT_RIGHTCLICK	This action indicates that the cursor routing keys should simulate right mouse

	clicks.
SAMBRLACT_CONFIG	This brings up a dialog box for configuring the Braille display settings and features.
SAMBRLACT_LITERARY_BRAILLE	Toggles the braille display between computer and literary braille.
SAMBRLACT_TAB	Duplicates the keyboard TAB key
SAMBRLACT_SHIFT_TAB	Duplicates the keyboard Shift+TAB keys
SAMBRLACT_ENTER	Duplicates the keyboard Enter key
SAMBRLACT_ESCAPE	Duplicates the keyboard Escape key
SAMBRLACT_CURSOR_UP	Duplicates the keyboard cursor up key
SAMBRLACT_CURSOR_DOWN	Duplicates the keyboard cursor down key
SAMBRLACT_CURSOR_LEFT	Duplicates the keyboard cursor left key
SAMBRLACT_CURSOR_RIGHT	Duplicates the keyboard cursor right key
SAMBRLACT_BRAILLE_KEY_1	Braille entry key row 1 left
SAMBRLACT_BRAILLE_KEY_2	Braille entry key row 1 right
SAMBRLACT_BRAILLE_KEY_3	Braille entry key row 2 left
SAMBRLACT_BRAILLE_KEY_4	Braille entry key row 2 right
SAMBRLACT_BRAILLE_KEY_5	Braille entry key row 3 left
SAMBRLACT_BRAILLE_KEY_6	Braille entry key row 3 right
SAMBRLACT_BRAILLE_KEY_7	Braille entry key row 4 left
SAMBRLACT_BRAILLE_KEY_8	Braille entry key row 4 right
SAMBRLACT_BRAILLE_SPACE	Braille space key

3.33 SamGetButtonPressed

This function is used retrieve details of which button has been pressed on the receipt of a **SAMBRLBTN_STRIP** message.

```
DWORD SamGetButtonPressed(
    DWORD  devid,
    DWORD  queueid,
    DWORD *unit,
    DWORD *strip,
    BYTE  *pbutton,
    DWORD *size
    );
```

Parameters

devid

The wParam value from the message.

queueid

The lParam value from the message.

strip

This points to a DWORD to store the returned strip number for the button press.

pbutton

This points to buffer of size bytes to hold returned button combination. If strip is of type **SAMSTRIP_KEYS** this will be byte array forming a bit mask, indicating which button(s) are pressed. For other strip types it will be a **BRAILLEBUTTONPRESS** structure.

size

A pointer to size of the buffer at pbutton, updated with the size required on exit.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_UNIT – devid does not contain a valid unit number.

SAMERROR_INVALID_VAL – queueid is not valid.

SAMERROR_INVALID_ADDRESS - The address pointed to by pbutton is not valid or does not have write access.

Remarks

When the driver detects that a button has been pressed it will send a message to the client and store the details of the button in a queue, allowing multiple presses to be handled before the client has responded. On receipt of the message, the client calls this function with the parameter values from the message to retrieve the details of the key press from the queue. Separate messages will be sent for each press.

If pbutton is NULL, only the required size of the buffer will be return in size.

See **BRAILLEBUTTONPRESS** for a description of the button information structure.

The following functions are from the SAM version 1 client API. They are implemented in version 2 for backwards compatibility with existing clients, but should not be used in any new code.

3.34 SamQueryBraille

This function gives information about each Braille unit. You can call **SamControl(SAMCONTROL_NUM_BRAILLE)** to determine how many units there are. Unit information can be requested at any time without first opening the unit or while the Braille display is in use.

```
DWORD SamQueryBraille(
    DWORD          unitid,
    BRAILLEPARAMS *lpunit,
    DWORD          langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

langid

This specifies the language that the Braille display description string is reported in. U.K. English (44) is guaranteed to be available.

Return Value

This function fills the parameter block pointed to by lpunit with the units parameters.

The function returns zero if it succeeded. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpunit)

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpunit is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

Remarks

See **BRAILLEPARAMS** for a description of the parameters structure.

3.35 SamSetDisplay

This function used to send text to the Braille display.

```
DWORD SamSetDisplay(
    HANDLE hBraille,   // unit identifier
    WORD   *dispdata,  // Dot pattern for main
                       // display area
    WORD   *statdata   // Dot pattern for
                       // status cells
    );
```

Parameters

hBraille

The handle to an open unit.

dispdata

This points to a block of WORDs representing the dot pattern to be displayed in the main area of the unit. If dispdata = NULL, the display cells will remain in their current state. Dispdata should contain the same number of WORDs as Braille cells as reported by **SamQueryBraille**.

statdata

As dispdata, this points to the dot pattern to be displayed on the status cells, and may be NULL to leave the cells in their current state.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_ADDRESS – the address pointed to by dispdata or statdata is not valid or doesn't have read access.

Remarks

The format of each WORD in dispdata and statdata is in a standard form as for **SameSetDisplayStrip**.

3.36 SamClearDisplay

This is used to clear the entire Braille display.

```
DWORD SamClearDisplay(
    HANDLE hBraille
    );
```

Parameters

hBraille

The handle to an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE - hBraille is not valid.

Remarks

All cells on the Braille display will be blanked (both cells in the main area and status cells). Any cursor will also be removed.

This call is equivalent to a call to **SamSetDisplay** and **SamSetCursor**.

3.37 SamSetCursor

This is used to position, set the appearance and blink rate of a cursor.

```
DWORD SamSetCursor(
    HANDLE hBraille,
    long   pos,
    WORD   shape,
    DWORD  rate
    );
```

Parameters

hBraille

The handle to an open unit.

Pos

The position of the cursor in the range 0 to the number of cells - 1, 0 is the leftmost cell in the display area. A value of -1 turns off the cursor.

Shape

The dot pattern of the cursor. When the cursor is “on”, this pattern is OR’d with the dot pattern of the cell under the cursor.

Rate

The blink rate of the cursor from 1 to the maxrate parameter as reported by **SamQueryBraille**. A value of 0 yields a static cursor. 1 is the slowest blink rate.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_VAL – pos or rate is out of range.

Remarks

The cursor will be repositioned and the display data unaffected.

A cursor may not be positioned in the status area.

The blink rate of the cursor will also affect any blinking text on the display.

See **SamSetDisplayStrip** for the format of shape.

3.38 SamQueryButton

This function is used to get a description of each button the Braille display has.

```
DWORD SamQueryButton(
    DWORD  unitid,
    DWORD  num,
    WCHAR *description,
    DWORD *size,
    DWORD  langid
    );
```

Parameters

unitid

The number of a Braille display unit.

Num

The number from 0 to the number of buttons -1 to get a description for.

Description

The address where to store the description, or NULL.

Size

The address to store the length of the description.

langid

The language code of the description.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_ADDRESS – if the address of description is invalid or does not have write access.

SAMERROR_INVALID_LANGID – if the language id is not valid.

Remarks

If description = NULL, this function returns the size of the description only.

This function can be used regardless of whether the unit is open or in use by another application.

This function returns the descriptions of the buttons that make up the standard buttons on the display.

3.39 SamIsKeyValid

This function determines if a particular key combination is valid for the unit.

```
BOOL SamIsKeyValid(
    DWORD unitid,
    DWORD key
    );
```

Parameters

unitid

The number of a Braille display unit.

Key

A bit mask indicating which of the keys is pressed. Bit 0 corresponds to button zero.

Return Values

This function returns TRUE if the unitid is valid and the button combination is valid.

Remarks

As there is only one class of buttons (apart from cursor routing buttons) there may be certain combinations of buttons that are not valid (for example a combination of thumb keys and other buttons).

3.40 SamGetButtonCombination

This function is used to obtain sensible default button combinations for a variety of client software Braille related features.

```
DWORD SamGetButtonCombination(
     DWORD  unitid,    // Unit number
     DWORD  action,    // Braille related action
     DWORD *button     // Returned combination
     );
```

Parameters

unitid

The unit number of a Braille unit.

action

One of the actions defined in ***brldacts.h***, see below.

button

This points to a DWORD to store the returned recommended default button combination.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_ADDRESS – the address pointed to by button is not valid or does not have write access.

SAMERROR_INVALID_VAL – there is no recommended default button combination.

SAMERROR_INVALID_UNIT – the unit number is not valid.

Remarks

The actions defined in ***brldacts.h*** and see **SamGetButtonCombinationStrip** for the associated meanings.

Note that it is likely that a particular display may not support all action codes (because, for example, there aren't enough physical keys or sensible key combinations). If the result is **SAMERROR_INVALID_VAL**, you should ignore the combination.

4. Structure Reference

This section documents the structures associated with SAM.

4.1 SYNTHPARAMS

SYNTHPARAMS give information about a selected synthesiser unit.

```
struct {
    WCHAR description[128];
    char  identifier[16];
    WORD  params;
    DWORD caps;
    DWORD langs;
    DWORD voices;
} SYNTHPARAMS;
```

Members

description

This is a description of the device. Use this string to describe the device if you present the user with a choice of units. The description string will be in the language you requested when you called **SamQuerySynth**.

It will contain a consistent unique description of the device, and include its connection. This allows you to use this string to 'remember' which synthesiser the user has selected so you can match it to the list of available synthesisers when your application is next started.

The recommended format is "manufacturer, device name, connection".

E.g.

"Dolphin Apollo 2 on COM 1",

"DecTalk Internal",

"KeyNote Gold PC on port 954",

"Artic Transport on COM 2".

identifier

This is a string (maximum 15 characters and the zero terminator). This will be set by the driver to a unique string that identifies the device. The purpose of this string is to allow you to match parameter sets with other devices. Where synthesisers have a common command set but a number of different connection methods (e.g. serial, ISA card, PCMCIA etc.), then the device may have several drivers, one for each type of connection. Each driver will have the same identifier string if the parameter sets are identical.

params

This specifies the number of parameters the device has. Each parameter can be queried with **SamQueryParam.**

Parameters are numbered from zero to params-1.

caps

This is a bit field, which specifies the capabilities of the device and other information. This information is static and will never change for any given synthesiser.

The following flags are defined:

SAMCAPS_BATTERY – unit is powered by batteries

SAMCAPS_REMOVABLE – unit can be unplugged by the user (without powering down).

SAMCAPS_PHRASEINDEX – unit can do phrase indexing.

SAMCAPS_WORDINDEX – unit can do word indexing.

SAMCAPS_LETTERINDEX – unit can do indexing on an individual character basis.

SAMCAPS_TALKING – unit can report whether it is currently talking or has more to say in the call to **SamIndex().**

SAMCAPS_PHONEME – unit supports the standard ISO Unicode phoneme character set.

SAMCAPS_HARDWARE – unit is a hardware synthesiser.

SAMCAPS_SOFTWARE – unit is a software synthesiser (uses a sound card).

SAMCAPS_RAWOUTPUT – the driver can send raw data to the synth.

SAMCAPS_SPEAKINCONTEXT – the synthesiser can perform context sensitive prounciation by procssing addioniton (unspoken) text.

SAMCAPS_SPELLPHONETIC, SYNTEXT_SPELLATC – in addition to the default pronunciation of letters in spelling mode, the synthesiser can use phonetic letter sounds, and international Air Traffic Control letter names.

langs

The number of different languages for which description strings are available (**SamQueryParamDesc** and **SamQueryParamChoice**).

Use **SamGetLangId** to get available Language id's.

This is nothing to do with which languages can be spoken.

voices

This specifies the number of preset voices available. Each voice is numbered from zero and can be retrieved using **SamGetVoice**. This value will be 1 or greater. (there are never zero voices!)

Remarks

The description string is stored in Unicode.

4.2 SAMPARAM

The **SAMPARAM** structure is used to specify a single synthesiser parameter. You retrieve information about a parameter using **SamQueryParam**.

```
Struct {
    WORD  type;      // type of parameter
    DWORD range;     // number of values allowed
    long  first;     // offset value for user
    WORD  id;        // parameter id value.
} SAMPARAM;
```

Members

type

This specifies the type of parameter. There are several types listed below. These are mutually exclusive.

SAMPARAM_NUMERIC – the parameter is numeric. range specifies the number of different values. The valid range of values is zero to range-1. This is normally used for numeric parameters such as Speed, Pitch, and Volume.

SAMPARAM_CHOICE – this parameter is a multi-choice parameter. This means that each value in the range is not necessarily related to any other.

Multi-choice values should be used for any parameters that are not numeric, but can be represented numerically, such as Gender, Speaker Table or Voice. range specifies the number of different values. The valid range of values is zero to range-1.

To get a description of each possible value to present to the user you should call **SamQueryParamChoice**.

SAMPARAM_COMPOUND – this type of parameter is a mixture of the above. range specifies the number, but the valid numeric values are not contiguous (or even ordered).

This type of parameter should be used where the meaning of a value may change depending on the users configuration of the synthesiser and you want to save the parameter to disk. For example, language. If the synthesiser has the capability of changing languages, then a simple number is no good.

Use **SamQueryParamValue** to get the numeric value for each possible value from 0 to range-1. Use **SamQueryParamChoice** to get a description of each possible value.

In addition, the following flags are defined. These are inclusive.

SAMPARAM_INPHRASE – if this bit is set, then this parameter can be used in mid phrase without causing the synthesiser to take a breath (or the driver putting a comma or carriage return in.) This is for advisement only. The driver should not fail if parameters are changed in the middle of a phrase, but it may have to break the phrase to perform the parameter change.

SAMPARAM_DEFAULT – this indicates that this parameter can be set to a 'default' value, which means that the driver should use an appropriate value depending on the current preset voice setting.

range

This specifies the number of values for this parameter. If this value is 1, the parameter may be a 'required' parameter such as language (in a single language synthesiser) and the client can ignore it.

first

In this driver, all numeric parameters are specified from zero to range-1. When you present the values to the user, you should add the value of first. This only applies to **SAMPARAM_NUMERIC** parameters. For all other parameter types you should use the string obtained with **SamQueryParamChoice.** To simplify your code you can call **SamQueryParamChoice** with numeric parameters and the driver will add first on automatically and generate the appropriate decimal string. However, no formatting or justification will be supplied.

Beware that first might be negative.

id

The id provides a way for you to distinguish particular parameters that are common between different synthesisers, such as Volume, Speed and Pitch. All of the common synthesiser parameters have a defined id value.

If the synthesiser does not support a parameter then when you call **SamQueryParam** for each parameter, none of its parameters will have that id value. If the synthesiser supports parameters in addition to the list of particular id's then those parameters will have the id value of **SAMID_UNKNOWN**.

Remarks

The id values defined are:

SAMID_VOLUME (numeric) – The Volume or Amplitude of the speech. Low values are quieter, higher values are louder.

SAMID_SPEED (numeric) – The overall speed of the speech. The higher the value the faster the speed.

SAMID_PITCH (numeric) – The pitch of the speech. The higher the value the higher the frequency of the speech.

SAMID_PROSODY (numeric) – The excited-ness or flatness of the speech - how much the pitch varies with time over the phrase. A low value should produce monotone speech (like a Darlek). A high value should indicate more expressive speech.

SAMID_WORDPAUSE (numeric) – The pause between each word. Higher values indicate longer pauses.

SAMID_PHRASEPAUSE (numeric) – The pause between each phrase or sentence. Higher values indicate longer pauses.

SAMID_LANGUAGE (compound) – The language of the speech. The actual numeric values of this parameter retrieved using **SamQueryParamValue** will match the language codes specified in ***sam.h***.

There may be more id values defined. Please check ***sam.h*** for additional codes.

4.3 SPECIALCHAR

This is used to specify a special character in a call to **SamGetCharSet**.

```
struct specialchar{
    WCHAR uc;          // unicode character
    WORD  padding;     // for structure alignment
    DWORD meaning;     // meaning value
} SPECIALCHAR;
```

Members

uc

The Unicode value being defined.

padding

Unused. Makes sure structure is memory aligned.

meaning

The meaning value of this unicode character. See the sections on Special characters for more information.

4.4 BRAILLEPARAMSEX

This structure contains information about a Braille display unit and is filled in by **SamQueryBrailleEx**.

```
Struct brailleparams
{
    WCHAR description[128];
    char  identifier[16];
    DWORD strips;
    DWORD maxrate;
    DWORD caps;
    DWORD langs;
} BRAILLEPARAMSEX;
```

Members

The description, identifier and langs members behave as their **SYNTHPARAMS** counterparts.

strips

The number of strips that the device supports. The properties of each strip can be retrieved using **SamQueryStrip**.

Maxrate

This contains the highest valid value for the rate parameter in a call to **SamSetCursorStrip**. This value defines the range of speeds for blinking characters.

caps

This is a bit mask indicating what features a unit possesses:

SAMBRLCAPS_BATTERY – the device is battery powered.

4.5 STRIPPARAMS

This structure contains information about a Braille strip and is filled in by **SamQueryStrip**.

```
Struct stripparams
{
    WCHAR description[128];
    DWORD type;
    DWORD length;
    DWORD buttoncombs;
    DWORD buttonstate;
    DWORD orientation;
    DWORD caps;
} STRIPPARAMS;
```

Members

description

A unicode string describing the purpose of the strip.

type

The type field contains one of the following values that determine the function of the strip.

SAMSTRIP_DISPLAY – the strip consists of length number of main display cells, and buttoncombs routing buttons per cell. A device with more than one row of display cells will return multiple display strips.

SAMSTRIP_STATUS – the strip consists of length number of status display cells, and buttoncombs status routing buttons per cell. A device with more than one row of status cells will return multiple status strips.

SAMSTRIP_AUXILARY – the strip consists of length number of auxiliary cells (not display or status cells), and buttoncombs routing buttons for each cell. Multiple auxiliary strips are not considered as multi-row.

SAMSTRIP_BUTTONS – the strip consists of length number of buttons. Only one button in the strip can be pressed at any one time, but the code returned can be augmented by shifting keys, see the description of buttoncombs. This strip type also supports other discreet 1D and 2D positional value returning devices such as sliders, dials and tracker pads.

SAMSTRIP_KEYS – the strip consists of length number of general-purpose keys. These differ from buttons in other strip types, in that a combination of pressed keys can be returned rather than a single button pressed value. The keys need not exist in an arrangement of a single row as for the other strip types.

length

The length field gives the number of braille cells and/or buttons in the strip. Strips of type **SAMSTRIP_DISPLAY**, **SAMSTRIP_STATUS** and **SAMSTRIP_AUXILARY** have cells, and optionally routing buttons if buttoncombs is non-zero. Strips of type **SAMSTRIP_BUTTONS** and **SAMSTRIP_KEYS** have no cells, but this number of either single press buttons or multi purpose keys. Only one row of cells is supported in a strip, additional rows are described by subsequent strips.

buttoncombs

The buttoncombs field gives the number of button combinations. For strips including cells, this will normally be the number of routing buttons associated with each cell, i.e. one for just primary routing buttons, and two if there are auxiliary routing buttons. However if device can use shifting keys to augment the codes returned from each button, this field describes the number of combinations that are possible of both real buttons and shifted buttons.

Examples:

- If there is one routing button and one shift key, there are two button combinations.
- If there is one routing button and two shift keys, there may be 3 or 4 combinations depending on if both shifting keys can be used at once.
- If there are primary and auxiliary routing buttons and one shifting key, there are at least 4 combinations.

For devices without cells, the combinations can be used to reflect the actions of shifting keys on each physical button. In the case of strip types that represent two-dimensional input, the number of buttons acts as the X-axis position, and buttoncombs acts as the Y-axis position (or vice versa, depending on the orientation field).

buttonstate

The buttonstate field gives information on whether the devices supports button up/down state information. A value of zero indicates that only button pressed events can be returned. A value of one indicates that the device can return both button down and button up events. Some devices may support more than one state for the button, such as a joystick with multiple fire buttons, or a track pad which can return the pressure applied or detect double clicks, in which case the value is the maximum number of states that can be returned.

orientation

The orientation field describes the primary orientation of the strip, and consists of three values:-

SAMORIENT_NONE	(orientation not appropriate)
SAMORIENT_HORIZONTAL	
SAMORIENT_VERTICAL	

Where a display offers more than one strip of the same type and orientation, it will always return horizontal strips in top to bottom order, and vertical strips in left to right order. Cells and buttons on vertical strips are always numbered from top to bottom.

caps

This is a bit mask indicating the features a strip possesses:

SAMBRLCAPS_EIGHTDOT – the strip has eight dot cells (if un-set, display is six dot).

SAMBRLCAPS_CURSOR – This indicates that a representation of cursor is possible for this strip and the **SamSetCursorStrip** function may be used.

SAMBRLCAPS_SLIDER –Indicates for strips of type **SAMSTRIP_BUTTONS** that this is a slider, which can return buttons number of discreet positions. These may be augmented by shifting keys to return buttoncombs combinations per position. The orientation field describes if the slider is horizontal or vertical. The slider can be moved with or without any associated button by setting the buttonstate field to 2, and returning button up events for movement without the button pressed, and button down events for movement with the button pressed.

SAMBRLCAPS_DIAL - Indicates for strips of type **SAMSTRIP_BUTTONS** that this is a rotary dial which can return buttons number of discreet positions (similar to a volume control). These may be augmented by shifting keys to return buttoncombs combinations per position. The orientation field describes if the dial represents horizontal or vertical movement.

SAMBRLCAPS_WHEEL – Indicates for strips of type **SAMSTRIP_BUTTONS** that this is a wheel type control with buttons set to 3, indicating the wheel is not being turned (**SAMWHEEL_NOTRUN**), turned left/down (**SAMWHEEL_TURNL**/**SAMWHEEL_TURND**), or right/up (**SAMWHEEL_TRUNR**/**SAMWHEEL_TRUNU**) respectively. If the wheel can return rate of turn information, buttoncombs will be set to the number of discreet rates supported. The orientation field describes if the wheel represents horizontal or vertical movement. The not turning position allows button up/down pressed event for any associated button, to be generated when the dial is not being turned.

SAMBRLCAPS_2D – Indicates for strips of type **SAMSTRIP_BUTTONS** that this is a two dimensional input device (such as a track pad or joystick) which can return buttons number of discreet positions on the primary axis, and buttoncombs number of discreet positions on the secondary axis. The orientation field describes if the primary axis is horizontal or vertical.

SAMBRLCAPS_PRESSURE – For a 2D input device, this indicates that the device is capable of returning a value related to the pressure applied to the device. The buttonstate field gives the number of discreet values that can be returned.

4.6 BRAILLEBUTTONPRESS

This structure contains information about a button press returned by the **SamGetButtonCombinationStrip** and **SamGetButtonPressed** functions.

```
Struct
{
   DWORD button;   // Button number
   DWORD comb;     // Augmented value
   DWORD pressed; // Button pressed information
} BRAILLEBUTTONPRESS;
```

Members

button

The button field contains the number of the button, from zero to one less than the number of buttons in the strip. For routing buttons on strips with cells the button number corresponds to the cell number.

comb

The comb field contains which combination of real or shifted buttons has been pressed, from zero to one less than the number of button combinations per button. On a 2D input devices button will contain the value of the primary axis and comb will contain the value of the secondary axis.

pressed

The pressed field can contain information on button up and button down states. Drivers are only required to provide button down information, which will be indicated by **SAMBUTTON_DOWN**. A slider, rotary, dial or 2D device that can move without a button being pressed can indicate this by a value of **SAMBUTTON_UP**. The value may be used to indicate additional button states, such as the pressure applied to a track pad, up to but not including the number of pressed states returned by **SamQueryStrip.**

The following structure is from the SAM version 1 API, it is implemented in version 2 for backwards compatibility with existing clients, but should not be used in any new code.

4.7 BRAILLEPARAMS

This structure contains information about a Braille display unit and is filled in by **SamQueryBraille**.

```
struct {
    WCHAR description[128]; // description
    char  identifier[16];   // identifier
    DWORD width, height;    // dimensions
    DWORD status;           // No. of status cells
    DWORD buttons;          // Number of buttons
    DWORD maxrate;          // Maximum cursor
                            // blink rate
    DWORD caps;             // flags
    DWORD langs;            // number of languages
} BRAILLEPARAMS;
```

Members

The description, identifier and langs members behave as their **SYNTHPARAMS** counterparts.

width,height

The width and height members define the dimensions of the main display area of the Braille display. Data sent to the unit must conform to this length.

status

Status defines the number of status cells the display possesses.

buttons

Buttons defines how many buttons the display has. This number excludes the special types listed in the caps member.

Maxrate

This contains the highest valid value for the rate parameter in a call to **SamSetCursor**. This value defines the range of speeds for blinking characters.

caps

This is a bit mask indicating the features a unit possesses:

SAMBRLCAPS_ROUTE display possesses cursor routing buttons over display area.

SAMBRLCAPS_STATROUTE display has status cell cursor routing buttons.

SAMBRLCAPS_AUXROUTE display has auxiliary routing buttons over main display.

SAMBRLCAPS_AUXSTATROUTE display has auxiliary routing buttons over status area.

SAMBRLCAPS_EIGHTDOT display has eight dot cells (if un-set, display is six dot).

5. The SAM Device Driver

The SAM device driver is a 32-bit DLL that provides an interface between SAM and the synthesiser or Braille display hardware. It contains a set of API functions that provide device-independence. The API set is documented in the Function Reference below.

The API set contained in this driver is not intended to be called directly by any application. It is used solely by the Synthesiser Access Manager (SAM). SAM controls access to each device and passes SAM API calls to the device driver as appropriate. Many of the function calls documented in this driver are very similar to the client API provided by SAM. SAM is just an access manager and server process.

In order to be compatible with SAM a device driver must implement this API. All API functions are required depending on the type of driver, however the individual capabilities of each synthesiser or driver may be different.

Speech synthesiser drivers must implement all the functions beginning with **Synd**. Braille display drivers must implement all the functions beginning with **Brld**.

Combination devices that consist of a single piece of hardware containing both a synthesiser and braille display must implement all functions.

5.1 Design Aims

The basic design aim of this device driver API is to provide the following.

- Complete hardware device independence.
- Assurance that the same actual spoken words will be uttered regardless of the actual device used.
- Consistency of speech across all devices.
- Identical API for Windows NT so talking software will not require any modifications.
- Fully multi-lingual using UNICODE to avoid code page problems.
- Support for Plug 'n' play speech devices and hot-docking synthesisers.
- Provides automatic synthesiser and braille display detection where possible.

5.2 Device Independence

The driver provides a common API regardless of the actual hardware. The driver can describe the capabilities of the device to the client. The client does not need to be aware of the actual physical connection to the device.

Functions are provided to report the actual set of parameters, the character set and punctuation information to the client. The client is therefore able to ensure that it can be sure of exactly which actual words are spoken, regardless of the physical device. This is important for dealing with semi-intelligent devices that perform context dependent processing. It also makes client speech software behave consistently (in terms of punctuation) across a range of hardware.

5.3 Extendible

The driver enumerates all the available parameters and ranges to the client. Parameters are provided as an extendible list so future hardware can add new parameters and existing software will be able to use them. This also allows software to use the full set of parameters available on different synthesisers instead of being restricted to a common subset.

5.4 Multi-Lingual

The driver must be able to produce a description for each parameter. The client can enumerate the languages for which descriptions are available.

All text is represented and transferred in Unicode to avoid problems with code-pages and the limited range of only 256 available values per character.

5.5 Protection

Each unit can only be accessed by one client at a time. Any sharing of devices is handled by SAM at a higher level. This simplifies the code required in each driver. The driver should generate handles when units are opened, and fail any subsequent calls that don't have the correct handle.

SAM is responsible for loading and unloading drivers. The 32-bit DLL is loaded into the address space of SAM so if there is a fault, the screen access software should not be affected.

5.6 Threads and Recursion

If the device driver only supports one unit at a time then all calls to the driver will be from a single thread. Function calls will not overlap or need to be re-entrant. The driver can assume that whilst it is processing a call to one of these functions it will not be pre-empted and called again from a different thread to another function.

If a driver supports more than one unit of the same type simultaneously then it may be called by two or more threads at the same time (one for each unit) and will have to maintain its own per-unit or per-thread data. The same thread will be used for each unit whilst the unit is open.

Internally SAM creates a thread to service each device unit. This means that if a driver thread becomes busy, other processes and threads will not be affected.

The functions **SyndInitialise**, **BrldInitialise**, **BrldCloseDown** and **SyndCloseDown** will be called by SAM's main thread. **SyndControl(SAMCONTROL_NUM_SYNTH)** and **BrldControl(SAMCONTROL_NUM_BRAILLE)** will also be called by SAM's main thread.

If the driver needs to perform extensive background processing to manage the device it should create its own thread. Function calls to the driver should return as quickly as possible.

5.7 Memory Model

The driver should be implemented as a WIN32 Dynamic Link Library. It should export all of the functions specified by name. SAM will perform the loading and freeing of the driver as it requires.

The driver DLL will only be loaded into a single instance of a process at one time so all addresses are 32-bit near pointers into the current process address space.

If required the driver may use any other modules it requires, such as a VxD or it can thunk to a 16-bit DLL.

The driver must clean up any memory and objects it has allocated when it is unloaded.

The driver should work under Windows NT.

5.8 Driver Installation

Each driver will be stored in its own directory. Any additional support files that it uses must also be stored in that directory (or any subdirectory).

SAM uses SDI files to identify which DLL is the driver. This file is in the standard Windows .INI format and contains one section with two values: the name of the driver (used in the SAM configuration dialog) and the name of the driver DLL.

```
[SAM Driver]
name=Dolphin Apollo
driver=DOLSER32.DLL
```

5.9 Function Exporting

Each driver must export the full set of functions using the standard WINAPI C Calling convention (the same as the Win32 API). Speech drivers must only export the **Synd...** functions. Braille drivers must export the **Brld...** functions, however version 2 drivers can omit the version 1 only functions, as SAM will call the new API in preference. Combination drivers must export both sets of functions. SAM uses **GetProcAddress** on the driver and determines if it is a speech, braille or combination driver depending on the set of functions it exports, it also uses this to detect the presence for version 2 **Brld...** functions,

5.10 Combination Devices

Combination device drivers (speech and braille) will be treated by SAM as two separate drivers (as if there were two different DLL files). This means that both **SyndInitialise** and **BrldInitialise** (and other similar functions) will be called in the single driver. In addition, the driver should report two units, one synth unit and one Braille unit.

However, to keep the driver code simple, SAM will only create one thread to service both devices for identical unit numbers.

For example, a combination speech and Braille device (e.g. the Tieman CombiBraille) will report 1 speech unit and 1 Braille unit. SAM will open and close each unit separately. It is up to the driver to deal with the single communications channel to the physical device. SAM will only create one thread to service both units, so the driver will not be pre-empted by a call into the other unit, and therefore will not have to contain any thread locking semaphores to serialise access to its communications routines.

Calls to **SyndControl** and **BrldControl** are handled slightly differently. A combination device must export both of these functions, but only one will ever be called (which one is undefined). They should therefore both do the same thing (simply have one call the other in the driver!). If there are specific settings for Braille and speech, a property sheet could be used.

5.11 Braille Strip Organisation

Braille device drivers must implement at least on strip of type **SAMSTRIP_DISPLAY**. If there is only one physical strip of cells, the driver should offer configuration options to allow the user to choose if it should be returned as one display strip or split in to a display strip and status strip at given position. Devices with a gap between status and display cells should return separate display and status strips.

If the device has more than one row of display cells, these should be returned as separate **SAMSTRIP_DISPLAY** strips. They should each be of the same length to allow use from the version 1 SAM API which can only handle displays with the same number of cells in each row. If the rows are of different lengths, only the first should be a display strip and others should be returned as strips of type **SAMSTRIP_AUXILARY**. Similarly for multiple rows of status cells, strips of type **SAMSTRIP_STATUS** should be returned, with an equal number of cells in each.

If there are multiple strips, horizontally orientated strips should be returned in top to bottom order, and vertically orientated strips in left to right order. Cells in a horizontal strip have index zero on the left, and cells in a vertical strip have index zero at the top.

If a strip with cells has routing buttons, the button numbers generated will correspond to the cell index of the cell it is over. If there are more buttons than cells, for example an extra button over a gap between a status and display strips, this should be returned in a separate strip of type **SAMSTRIP_BUTTONS**.

Drivers should avoid changing the number or order of strips that they provide as programs may persistently store the strip and button pressed information for actions returned by **SamGetButtonCombinationStrip()**. If the strip indexes change, this information will be invalidated. Therefore it is suggested that any optional strips be returned last. If this is not possible, the driver should return a different identifier string in **BRAILLEPARAMSEX** for each configuration with variable strip numbers/types.

5.12 Supporting Old Braille Driver API’s

Drivers should be written to support the new strip based API functions, they do not need to implement those functions from SAM version 1 that are labelled as having been superseded. SAM will automatically convert the new strip information in to a form the version 1 client applications understand.

However, it is be possible for a driver to implement both sets of APIs, in which case programs using version 2 SAM client API’s will cause the version 2 driver API’s to be used, and programs using version 1 SAM client API’s will call the version 1 driver API’s.

This could be done in order to present new features in a backwards compatible way. For example if a new driver returns strips of type **SAM_STRIPBUTTONS** with **caps** of **SAMBRLCAPS_WHEEL** for a wheel control, when used with old clients SAM will ignore this strip. However, if the driver implements the old API it could choose to implement the wheel direction inputs as two buttons inputs. I.e. An additional two **buttons** in the buttons field of **BRAILEPARAMS** and returning button pressed events of **SAMBRLBTN_NORMAL**.

If a driver does implement both sets of API's, the information delivered to the **BUTTONFUNC**/**BUTTONFUNCSTRIP** callback function given in **BrldOpenBraille** will depend on the type of the current client. The callback function is actually the same with 3 DWORD parameters, but interprets these appropriately.

If the driver receives a call to **BrldQueryBraille** it being used from a version 1 client and so should subsequently issue the callback with parameters of devid, button and type, where type is one of the following; **SAMBRLBTN_NORMAL, SAMBRLBTN_CSRDISP, SAMBRLBTN_CSRSTAT, SAMBRLBTN_AUXDISP, SAMBRLBTN_AUXSTAT**.

If the driver receives a call to **BrldQueryBrailleEx** it is being used from a version 2 client, and should subsequently issue the call back with parameters of devid, pbutton and strip.

6. SAM Synthesiser Device Driver Programming Overview

This is a general overview of how the client uses the driver to access a synthesiser. Normally the client will be SAM, but SAM is just passing the speech requests directly from the talking application or screen reader.

6.1 Driver loading

When the driver DLL has been loaded, SAM calls **SyndInitialise()** once only to give the driver its registry key location so it can check its configuration.

6.2 Units

Each driver DLL may support zero or more units.

When the driver is loaded, it should do any internal initialisation that doesn't involve a synthesiser.

When **SyndControl(SAMCONTROL_NUM_SYNTH)** is called the driver must return how many units have been configured (if any). SAM may then call **SyndOpenSynth** to access a unit.

It may be possible for the driver to be loaded without a synthesiser plugged in or turned on. In this case **SyndOpenSynth** should return an error code, so a screen reader may produce an audible message indicating that no synthesiser can be found and the user will be prompted to check their synthesiser.

If SAM is unable to locate a single working synthesiser, it will scan all drivers and call then with **SyndControl(SYND_DETECT)** to attempt to locate a synthesiser. In this way, if the user moves or changes their synthesiser their talking software will be able to automatically use the new synthesiser.

In most cases, each driver will support only one synthesiser.

If the driver is capable of detecting a synthesiser on one of several ports then it should do so during a call to **SyndControl(SYND_DETECT)** providing this will not mess up any other hardware attached to the machine.

If a synthesiser is not configured or cannot be detected at the correct location, the driver should return zero for a call to **SyndControl(SAMCONTROL_NUM_SYNTH)**.

When the driver is unloaded, (in the call to **DLLMain**) it should shut down and mute any synthesiser. Some synthesisers can be powered down at this point. Otherwise, the driver should power-down the synthesiser in the call to **SyndCloseSynth**.

6.3 Opening and Closing

In order to validate access to the driver only one thread is allowed to open a device at one time. When a unit is opened it will not be opened again.

When a unit is closed using **SyndCloseSynth,** functions that produce speech should fail.

Normally SAM will open each device as requested by each client. If two clients want to use the same device, SAM will provide the sharing interface and generate new handles for each client. The driver does not need to concern itself with this.

There is no need for unit handles because the SAM driver will only ever be called from SAM. **SyndOpenSynth** and **SyndCloseSynth** are only really useful to allow the driver to turn the device on and off.

6.4 Unit Capabilities

The client can query the capabilities of the synthesiser device. It can do this before opening the device. API functions such as **SyndQuerySynth, SyndQueryParam, SyndQueryParamDesc, SyndQueryParamValue, SyndQueryParamChoice** and **SyndGetLangId** may be called to get this information.

To find out information about each unit the client should call **SyndQuerySynth**. This will fill a **SYNTHPARAMS** structure with information.

The unit parameters structure is of fixed size and format so the driver simply can copy a piece of static data when it is requested for it.

```
struct {
    WCHAR description[128];    // unit text description
    WORD params;               // number of parameters
    DWORD caps;                // unit capabilities
    WORD langs;                // number of languages
} SYNTHPARAMS;
```

See **SYNTHPARAMS** in the structure reference for a detailed description of each parameter.

Each synthesiser accepts a different character set, handles punctuation differently and has many different parameters. Using the query functions, a client can determine what the synthesiser will or won't do.

6.5 Speech Parameters

Each synthesiser supports a different set of speech parameters. Each parameter varies between devices in its meaning and possible range of values.

The client should use the various query functions to determine the set and meaning of each parameter.

First, **SyndQuerySynth** will be called. This contains params that specifies how many parameters there are. The parameters can be reported by the driver in any order, but the order MUST match the order used in the Parameter block structure (see below).

For each parameter, the client will call **SyndQueryParam** to get information about the parameter. **SyndQueryParam** takes a unit number, a parameter number (numbered from zero) and a pointer to a buffer to receive the data. The client should allocate the required memory for the **SAMPARAM** structure.

Each parameter is described by a **SAMPARAM** structure.

```
Struct {
    WORD type;              // type of parameter
    DWORD range;            // number of values allowed
    long first;             // offset value for user
    WORD id;                // parameter id value.
} SAMPARAM;
```

Each parameter has an id value. This value contains an identifier that lets the client determine the meaning of the parameter for the more common types (e.g. volume, speed and pitch). In addition, some other parameters such as punctuation and spelling must be numeric so the client can determine the meaning of some of the parameter values. (E.g. a punctuation value of zero *means* no punctuation).

There are three types of parameters: Numeric, Multi-choice and compound.

[Maybe we could add a 'string' type for a parameter to allow the user to enter the filename of an exceptions dictionary for example]

The type field specifies which type of parameter it is.

Numeric

Numeric values are used where a parameter represents a contiguous range of discrete values, such as speed, pitch or volume.

The numeric values must be contiguous and start at zero. The first offset value indicates to the client how to present the value to the user. To simplify client software the driver should provide a description for each numeric parameter as well.

Multi-Choice

A multi-choice value should be used where there is no sensible numeric value for each parameter. (E.g. "child" or "adult"). In addition, the driver must provide a description for each parameter.

Compound

A compound parameter is essentially a multi-choice parameter with a numeric value underneath. The numeric values do not have to be contiguous. This is important for a parameter such as language where different synthesisers may have different supported languages yet the client wants to be able to save a synthesiser configuration.

Flags

There are other flags defined for each parameter. These give additional information to the client. See **SAMPARAM** for a description of these flags.

6.6 Parameter Descriptions

The driver must be able to provide a description for each parameter and a description for each value for each parameter. The client will call **SyndQueryParamDesc** to get each parameter description and **SyndQueryParamChoice** to get a description for the parameter values.

All parameter strings should be stored in Unicode.

6.7 Speech API Functions

Some of the speech output functions pass data to the driver. This is passed by means of a pointer to a data block. While the thread of execution is in the driver, the data in the block will remain valid. When the function has returned the client may delete, invalidate or overwrite the data areas. Therefore, the driver must make a copy of any relevant data inside the API call (if it will need to access it after the call has returned).

The driver must never modify any of the data in buffers passed to it.

For functions that report back to the client, the driver must simply copy data to the address provided.

6.8 The Voice Block

When anything is spoken, the driver needs to know what voice (or what set of parameters to use) to speak in. Whenever the driver is called with text to speak, the client also passes a pointer to a voice block.

The voice block is a chunk of memory that contains an array of DWORD's, one for each synthesiser parameter.

The size of the voice block is therefore
`sizeof(DWORD)*`**`SYNTHPARAMS.params`**

The parameters should be stored in the voice block in the same order as they were enumerated using **SyndQueryParam**. The ordering of parameters is therefore entirely up to the driver.

The client must pass a complete voice block with each speech call every time so the driver is responsible for optimising output, i.e. only sending the changes to the synthesiser.

If a parameter is changed in mid-sentence and the synthesiser is not capable of performing such a change, the driver should break up the sentence into two.

A default voice block can be obtained along with the preset voices by calling **SyndGetVoice**.

6.9 Preset Voices

The driver can provide a number of preset voices. It must also provide a default set of parameters (a default parameter block). This is because the speech output functions require a voice block. The total number of voices must be specified in the **SYNTHPARAMS** structure.

The driver must provide at least one preset voice, number zero which is the default voice. Each preset voice is obtained by calling **SyndGetVoice** with a preset voice number.

Each voice parameter may be allowed to be set to a 'default' value. This tells the driver to use a default value that may be based on the setting of another parameter. If a parameter can be set to default the **SAMPARAM_DEFAULT** flag will be set in the type field of the **SAMPARAM** structure.

The parameters for volume, speed and language must not be allowed to be set to 'default' (the **SAMPARAM** structure for these parameters must not include **SAMPARAM_DEFAULT**).

The numeric value of 'default' is **SAMVAL_DEFAULT** (-1).

The client can call **SyndGetVoice** to get a copy of the voice block for any of the voices provided by the driver. This should specify default values for all the parameters that are not flagged as **SAMPARAM_DEFAULT**.

The client can call **SyndQueryParamChoice** to get a description of each voice.

6.10 Text to Speech Output

The most vital part of the speech driver is providing speech output.

There are three main functions used for generating and controlling speech output: **SyndMute, SyndAppend,** and **SyndSpeak.**

SyndMute immediately halts any speech in progress and clears any pending speech. The client can then call **SyndIndex** to obtain information about where the speech stopped.

SyndAppend is used to add words or phrases to the output buffer. This function should not cause the synthesiser to start speaking. The driver can start sending text to the synthesiser if it is sure that no speech will be generated.

SyndSpeak is called by the client to start speech. All previous text in calls to **SyndAppend** should commence to be spoken.

SyndSpeak implies a phrase ending. Calls to **SyndAppend** separated by a call to **SyndSpeak** should not be concatenated together into a single sentence, but a phrase ending should be placed between. This provides a method of terminating the phrase without needing to send a carriage return, full stop, etc. Also the synthesiser should cleanly finish the phrase without speaking any termination punctuation (e.g. period).

6.11 Chaining

If the synthesiser is speaking the client can send additional phrases to the synthesiser by calling **SyndAppend** and **SyndSpeak** without a call to **SyndMute.** The driver should then append the additional text onto the end of the current speech.

6.12 Indexing

The driver should provide indexing facilities so the client can determine the progress of the speech.

The driver should report the available indexing capabilities to the client in the **SYNTHPARAMS** structure so the client knows how accurate the information is likely to be. If the client requests indexing information at a greater resolution than the synthesiser can manage (e.g. the client tries to index by character, but the synthesiser can only manage word indexing) then the driver should report to the nearest index marker it can.

To perform indexing the client will cut up the source text into chunks. The driver should be able to report back to the client which 'block' is currently being spoken. This means that the client decides the actual index positions.

For example, if the client wants to speak a sentence and know which word is being spoken at any instance, it should call **SyndAppend** with each word individually with a unique *index* value.

To determine the index position the client calls **SyndIndex**. This function call should return quickly. (i.e. the driver should not actually spend the time asking the synthesiser (especially if it is connected via a serial interface!). The driver should retrieve the index count from the synthesiser asynchronously in the background.

The call to **SyndIndex** should return the index identifier for the currently speaking text block and other information.

Calls to **SyndAppend** should be concatenated together by the driver, so to be spoken as words, the client must still put a space between each word.

6.13 Punctuation and Spelling

To simplify the driver and allow the application to have maximum control of the speech, the synthesiser should generally have a fixed level of punctuation.

The driver must report in the call to **SyndGetCharSet** the set of characters that can be used to create a word (produce speech), the set of non-speaking modifier characters (such as apostrophe or comma) and the set of punctuation characters.

In addition, the driver must support a set of special characters that can be used for sound FX or pauses.

The driver must be able to spell all of its characters. The **SyndAppend** function contains a number of flags. If the spelling flag is set then the synthesiser should spell all of the characters in its input string. It will still only be called with characters in its character set. Spelling only applies to Alphabetic and modifier characters. Punctuation characters are always spelt. The synthesiser should not announce whether a letter is a capital or not. The synthesiser must be able to announce the standard letters a to z correctly as these are needed by screen readers to speak what is typed.

The Synthesiser should only speak what it is sent. It should not attempt any expansion of abbreviations or context sensitive processing.

6.14 Character Sets

The four sets of characters are known as the alphabetic, modifier, punctuation and special characters.

Only the alphabetic characters should be used to form words. Modifier characters should not produce any speech by themselves but only affect the pronunciation of alphabetic characters. The synthesiser must not attempt to perform context processing of the input string.

The punctuation characters should always be spelt by the synthesiser, even if they are in the middle of a word, i.e. they should break up the word.

These three sets of characters must be mutually exclusive.

To obtain the sets of characters the client will call **SyndGetCharSet** for each spoken language. A different set of characters is allowed per language.

The driver MUST supply character set information for each language that it reports that it can speak.

The driver should only take notice of characters in the speech string that are in its supported character set. Any other characters can be ignored. (The client shouldn't pass any characters that are not in the allowed character sets.

A typical character set might be:

Alphabetic: abcdefghijklmnopqrstuvwxyz0123456789

Modifier: , . ' - ! ?

Punctuation "£$%^&*()=+][{}#~@/;:

The space character (U+0020) must be part of the Modifier characters, because it does not produce any sound by itself.

Punctuation such as comma and full stop should be part of the Modifier set so that the client can decide if the synth should say the word 'comma' or put in a phrase pause by use of the spelling flag.

Numbers are part of the language because numbers should always be spoken as words, i.e. the synthesiser should form words from the sequences of digits in the same way as it forms words from sequences of letters. E.g. 123 should be spoken as "one hundred and twenty three" (or however it is pronounced in a given language). The driver should announce single digits if they are separated by a space or spelt.

The alphabetic character set does not have to contain both upper and lower case as both are equivalent when it comes to forming words, however the synthesiser should speak the same thing regardless of upper or lower case.

6.15 Phonetics

If the synthesiser supports phonetic characters then it should add the appropriate Unicode characters into its alphabetic character set. Unicode defines a set of standard codes that represent common phonetic symbols. (U+0250 to U+02A8)

There should be no special 'phonetic mode'. Any phonetic characters are simply part of the language character set.

6.16 Special Characters

Special characters can be put into the speech string to cause the synthesiser to perform a special action such as a slight pause or sound effect.

The client will call **SyndGetCharSet** to retrieve the set of special characters. The actual Unicode values for each special character is up to the driver providing there is no clash with the other characters sets reported in **SyndGetCharSet**, i.e. the code for a special character can be the same as a valid Unicode character code providing that the Unicode character is not included as one of the alphabetic, modifier or punctuation characters.

When the client calls **SyndGetCharSet** they get a set of **SPECIALCHAR** structures, which contain the Unicode value and the meaning. The meaning is encoded as a single 32-bit DWORD value.

Sound Effects

Sound effects should have a meaning value of between 0x80000000 and 0x80001000. It is irrelevant what the actual range of sounds that can be produced is because this is a speech device, not a music synthesiser!

The driver should ideally provide no more than 20 different sound effects.

Musical Notes

If the device is capable of playing musical tones then it should return meaning values of 0x810000aa where aa indicates the note number. There should be a progression of frequencies available where each increase in value represents one semitone. C4 has a value of 60 (3ch).

Therefore, C3 will be 48, E4 will be 64 etc.

Each musical time should be of approx 200ms duration. To play a longer note multiple codes will be concatenated together in the input string.

Pauses

Pause codes should have a meaning value of between 0x00000000 and 0x00001000 where the meaning value indicates the approximate pause length in milliseconds.

Telephone Tones

If Digital tone multi-frequency (DTMF) telephone tones can be generated by the synthesiser then they should have a meaning value from 0x00010000 to 0x0001000f (there are 16 different possible tones).

6.17 Raw Output

The driver should provide a facility to send raw data to the synthesiser. If it can it should set the **SAMCAPS_RAWOUTPUT** flag in the **synthparams.caps**. This facility is provided for people who want to fiddle with their synthesiser by embedding control codes in their word processor documents.

6.18 Errors

Hardware failure

If the hardware stops responding or is disconnected then any function may return **SAMERROR_UNIT_FAIL** to indicate to the client that the synthesiser has stopped without prior warning. The driver should keep attempting to communicate with the synthesiser (without taking up too much CPU time!) until it starts to respond again, or **SyndCloseSynth** is called.

API Parameter errors

Generally, the driver should check all parameters to its functions and return appropriate error codes. In the case where there is potentially more than one applicable error code, the driver should try to check the parameters in the order they are listed in the function arguments. Otherwise, it can return any applicable error.

6.19 Languages

Each driver should maintain two lists of languages.

One is a list of spoken languages this is needed for **SyndAppend** and **SyndGetCharSet**.

The other is a list of languages that descriptions are available for. This is obtained with the **SyndGetLangId** function and used with the various **SamQuery** functions.

Be careful that you don't mix up these two language lists.

To find out which languages a synthesiser can speak in, SAM will:

- Call **SyndQuerySynth** to find out the number of parameters.
- Repeatedly call **SyndQueryParam** to find which parameter specifies the language.
- Repeatedly call **SyndQueryParamValue** for the range of the language parameter to get the actual language id for each spoken language

To find out which languages descriptions are available in, SAM will:

- Call **SyndQuerySynth** to find out the number of languages.
- Repeatedly call **SyndGetLangId** to get each language id.

The meaning of the language id is the same for both lists (defined in ***sam.h***).

The driver MUST return character set information for each spoken language.

It does not have to return descriptions for each spoken language, but MUST at a minimum return descriptions in English.

Ideally, the driver will supply descriptions in any of the languages it can speak.

7. SAM Synthesiser Device Driver Function Reference

This section provides a function reference for all the synthesiser driver API functions. The structures are the same as defined in SAM.

7.1 SyndGetCharSet

This is called by the client to determine the set of characters that can be used to make up speech.

```
DWORD SyndGetCharSet(
    DWORD unitid,       // unit identifier
    DWORD charset,      // charset number
    DWORD *buffer,      // pointer to buffer
    DWORD *size,        // pointer to DWORD to
                        // receive size
    DWORD language      // requested language
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive).

charset

The identifier of the requested character set. This should be set to **SAMSET_ALPHABETIC, SAMSET_MODIFIER, SAMSET_PUNCTUATION** or **SAMSET_SPECIAL**, to request one of the four character sets.

buffer

This points to a section of memory where the information will be stored, (or it can be NULL). This memory must be allocated by the client before the function is called. The format and size of this is dependent on charset.

size

This must hold the address of a DWORD to receive the size of the required buffer. The driver must always store the size here, as well as storing the character set in buffer.

language

The requested language for the character set. This should be one of the available spoken languages for this unit.

Different languages can have different character sets.

Return value

The return value is zero if the function succeeded or an error code otherwise.

SAMERROR_INVALID_ADDRESS – if the address pointed to by *buffer* or *size* is not valid or doesn't have write access.

SAMERROR_INVALID_UNIT – the unit id value was invalid.

SAMERROR_INVALID_LANGID – the language id of the spoken language is not a valid language.

Remarks

If buffer is NULL, then the function stores the number of characters in the given character set in the location pointed to by size. This may be zero. The client should multiply this number by the size of each record to get the buffer size in bytes.

For alphabetic, modifier and punctuation character sets, each record consists of one Unicode value (stored as a WCHAR). The buffer will therefore contain a simple array of WCHAR's, one for each character.

Special characters are stored as an array of **SPECIALCHAR** structures.

The character set should be sorted into ascending order.

This function may be called with languages other than possible spoken languages. The driver may optionally return a character set, or can return **SAMERROR_INVALID_LANGID**.

7.2 SyndAppend

This is the main function used to send text to the driver.

```
DWORD SyndAppend(
    DWORD unitid,       // unit identifier
    WCHAR *text,        // pointer to unicode text
    DWORD length,       // number of characters
    DWORD index,        // index identifier
    DWORD *vblock,      // pointer to parameter
                        // block
    DWORD flags         // speech flags
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

text

A pointer to the start of the Unicode string which should be output. The string does not need to be zero terminated.

length

The number of characters in the text string. The size of the buffer pointed to by text must therefore be length`*sizeof(WCHAR)`.

To perform indexing the client can subdivide a source string into individual words and call **SyndAppend** with each word, without having to allocate or manipulate any additional memory.

There is no limit to the length of the text!

If length is zero, the string is assumed to be NULL terminated. (NULL meaning 2 bytes, both zero).

index

The index value is a unique value that is generated by the client to identify this piece of text. If the client performs indexing then this is the identifier that should be reported back to the client.

It is up to the client to ensure that the index identifier is unique for all pieces of text, unless it does not intend to do indexing, in which case index can be zero or some other constant value.

vblock

This is a pointer to a voice block for the speech. All of the text in this call to **SyndAppend** should use the speech parameters in vblock. This parameter may be NULL in which case the driver should use the same parameters as the last call to **SyndAppend** (or the default parameters if there has never been a call to **SyndAppend**).

Multiple calls to **SyndAppend** can contain the same address for vblock, however the driver must make a copy of any relevant data because the contents of *vblock may not be valid after the call has returned.

flags

The following flags are defined, which give additional information to the driver about how text should be processed:

SYNDTEXT_RAW – If this is set then the driver should send the text directly to the synthesiser without any pre-processing. This may be used by the client to send special synthesiser dependent commands. With the right screen reader software users will be able to embed synthesiser commands in their documents.

The driver should parse the string and interpret any synthesiser commands in the same way as the synthesiser will so the driver can keep track of the current state of the synthesiser.

SYNDTEXT_SPELL – If this is set then the driver should spell each character in this section of text. The characters received will still be restricted to the character set reported by the driver in **SyndGetCharSet**. All characters, including space must be spelt.

SYNTEXT_SPELLPHONETIC – This flags can be set in conjunction with the **SYNDTEXT_SPELL** flag to use the phonetic sounds for letters. For example in English “cat” would be “cuh, ah, tuh”.

SYNTEXT_SPELLATC – This flags can be set in conjunction with the **SYNDTEXT_SPELL** flag to use the international Air Traffic Control names for the letters, e.g. “cat” would be “charlie, alpha, tango”.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_UNIT – unitid is not valid, or the unit is not open.

SAMERROR_INVALID_ADDRESS – if the address pointed to by text is not valid or doesn't have read access.

Remarks

This function should only queue the speech output and not actually produce any speech. The client must call **SyndSpeak** before any speech can be generated.

The text string pointed to by *text* should only contain characters specified in the relevant character sets reported by the driver. Any characters not in these sets should be ignored by the driver.

All text strings in multiple calls to **SyndAppend** should be treated as a single phrase and concatenated together. The only reason the client will break up a sentence into multiple calls to **SyndAppend** is to perform indexing or to change a parameter in mid-sentence.

The client may call **SyndAppend** while speech is in progress. The driver should prepare to append the text onto the end of the current section of speech. It may be able to pre-send some text to the synthesiser in preparation for a call to **SyndSpeak**.

7.3 SyndMute

This is used to stop any speech in progress and clear any pending speech.

```
DWORD SyndMute(
    DWORD unitid
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_UNIT - unitid is not valid, or the unit is not open.

Remarks

This should halt speech immediately. The driver should be ready to report the index position that the speech reached with a call to **SyndIndex**.

This function may also be called after a number of calls to **SyndAppend** without a following **SyndSpeak.** In this case, the synthesiser should stop any speech and the buffers should be cleared.

7.4 SyndSpeak

This function is called to start the speech of text from one or more previous calls to **SyndAppend**. If possible the driver should have already sent the text to the synthesiser (without causing it to start speaking).

```
DWORD SyndSpeak(
    DWORD unitid,
    DWORD index,
    DWORD flags
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

index

The index value is a unique value that is generated by the client to identify the end of this speech block.

If the client requests the index marker and the speech has finished, the driver should return this value as the index.

flags

There are no flags defined so this parameter must be zero.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_UNIT - unitid is not valid, or the unit is not open.

Remarks

This function implies an end of phrase. The driver should make sure a phrase break occurs at the point this function is called. Therefore, any subsequent calls to **SyndAppend** are not directly concatenated onto the end of **SyndAppend** calls prior to this function.

7.5 SyndIndex

This function retrieves the index identifier for the currently speaking block of text and other information.

```
DWORD SyndIndex(
    DWORD unitid,
    DWORD *index, // address of DWORD to place
                  // index
    DWORD *flags  // address of DWORD for flags
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

index

This is a pointer to a DWORD. The driver should place the index identifier here. This is the value that was passed in the call to **SyndAppend** that the currently speaking text came from. If the speech has finished then this value should contain the value specified in the call to **SyndSpeak.** If the speech has not yet started (its a really slow device...) then it should return the index value of the first word.

If the synthesiser is unable to do word or phrase indexing then it should store 0 at this location.

flags

This is the address of a DWORD that the driver should set with the following flags as appropriate:

SYNDSPEECH_INPROGRESS – should be set if the synthesiser is speaking, or has more to say.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_UNIT – unitid is not valid, or the unit is not open.

7.6 SyndInitialise

This is called once only after the driver DLL has been loaded.

```
void SyndInitialise(
    CHAR *regkey,
    DWORD reserved
    );
```

Parameters

regkey

This is a pointer to a null terminated ANSI string, which contains the path to the registry key for this driver. The driver should make a copy of this string.

reserved

This can be ignored and should be set to zero.

7.7 SyndCloseDown

This is called once only to close down the driver DLL. The driver should close all open units and release any memory it has allocated, close any open files and perform any other necessary clean up tasks in preparation for unloading.

```
void SyndCloseDown(void);
```

Remarks

There are no parameters and no return value.

7.8 SyndControl

This function is used for a number of functions, all to do with detecting and configuring one or more units.

The Synthesiser driver allows for multiple units. Each unit is a distinct speech output device and is operated independently. Normally there will only be one unit unless the user has more than one synthesiser installed.

```
DWORD SyndControl(DWORD mode);
```

Parameters

mode

This will be one of the following values and indicates what the function should do.

SAMCONTROL_NUM_SYNTH

The function should return the number of synthesisers that it is configured for. It should not perform any detection or verification of devices. When the driver is manually configured or has auto-detected a synthesiser(s) it should save the physical location of the synthesiser(s) in the registry. It should return the number of synthesisers that was last detected or configured.

If there are no units configured (which will be true the first time the driver is ever used) then this function should return zero.

Return value

The number of units configured should be returned.

SAMCONTROL_DETECT

When SAM is installed for the first time there will be no synthesisers configured. SAM will call this function in each driver to attempt to detect an initial speech synthesiser.

The driver should perform any hardware detection it can to locate available synthesisers.

If the synthesiser is a plug 'n' play device then the Windows 95 device enumerator will have already found it and this driver can just query the registry as appropriate. If the synthesiser hardware has a proper Windows VxD driver that performs device detection, then again the registry can just be queried.

Otherwise, a synthesiser should be searched for providing the search procedure is safe and does not break any other hardware. Providing the Win32 API is used to access hardware, this should not be a problem.

Regardless of how many synthesisers are found (if any) the driver should save any relevant information in the registry so that subsequent calls to **SyndControl(SAMCONTROL_NUM_SYNTH)** will be accurate.

This function may also be called at the users request from the SAM control panel.

This function may be called whilst one or more units are open. In this case the driver can assume that those units are present and correct and only search for synthesisers at other locations.

If the driver only supports one unit at a time then it should halt when it locates the first one.

Return value

SYND_DETECT_OK – should be returned if the detection sequence completed and the driver can verify the presence (or lack) of a synthesiser.

SYND_DETECT_CANT – is the return code if the driver is unable to reliably detect a synthesiser. SAM may subsequently call **SyndControl(SAMCONTROL_CONFIG)** later.

SAMCONTROL_CONFIG

This function gives the driver the opportunity to allow the user to enter any configuration information that the driver needs. SAM will call this function if the user selects **CONFIG** from the SAM control panel.

This function should display a dialog box with any relevant options in. The dialog box should contain the connection location of the synthesiser. If the driver can auto detect a synthesiser then the box should contain 2 radio buttons for selecting between auto and manual configuration.

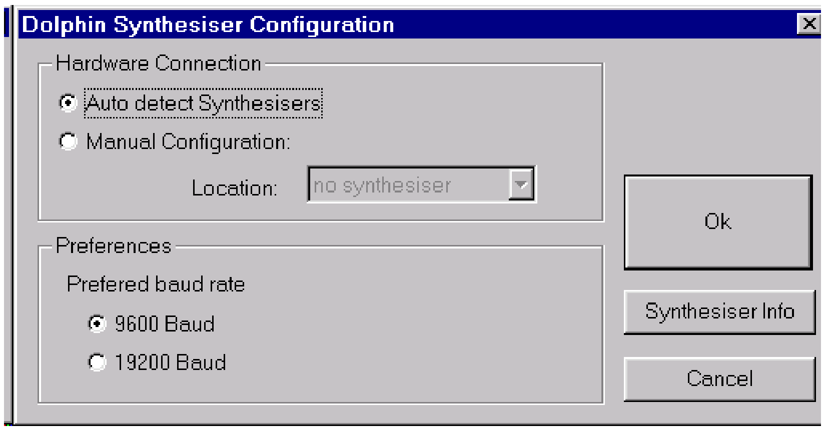
Any additional options are up to the driver. It could contain controls to adjust exceptions dictionaries, adjust languages etc.

This function should not return to SAM until the dialog box is closed.

Any changes to the actual synthesiser configuration should not be made until the driver is closed and opened again (with a call to **SyndInitialise**) regardless of any changes made. The dialog box should contain two buttons, **Ok** and **Cancel**.

There may be a unit in use whilst this function is called (so that a screen reader can make it speak). When this function returns SAM may close all units and then call **SyndCloseDown** followed by **SyndInitialise**.

Example


Dolphin Synthesiser Configuration

Hardware Connection

[✓ Auto detect Synthesisers]

Manual Configuration:

Location: no synthesiser

Preferences

Prefered baud rate

[✓ 9600 Baud]

19200 Baud

Return value

The return value tells SAM what to do next:

SAMCONFIG_CANCEL – if the user clicked on Cancel. SAM will do nothing.

SAMCONFIG_RELOAD – if a change has occurred that does not affect the location of the synthesiser, e.g. baud rate, exceptions dictionary. SAM will automatically close the driver with **SyndCloseDown** and re-open the driver with **SyndInitialise**. It will then notify applications of a driver configuration change.

SAMCONFIG_REDETECT – if a change affects the location (or existence) of a synthesiser, e.g. COM port location. SAM will then inform the user that changes will not take effect until the 'Detect All Devices Now' button is pressed. This will cause all drivers to be closed and re-started and the standard detection sequence to be initialised.

7.9 SyndOpenSynth

SyndOpenSynth is used to open a speech unit. Only one process can open a unit at a time.

```
DWORD SyndOpenSynth (
    DWORD unitid  // unit number
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive).

Return Value

The function returns zero if it succeeded.

If the unit failed to open the return value is one of the following.

SAMERROR_INVALID_UNIT - the unitid value was invalid.

SAMERROR_UNIT_OPEN - the unit is already open.

SAMERROR_UNIT_FAIL - the physical device is not responding, turned off, unplugged or otherwise unusable.

Remarks

The client will call **SyndCloseSynth** to release ownership of the device.

Only one client is allowed to open each unit at one time. You must return **SAMERROR_UNIT_OPEN** for any subsequent attempts to open a unit that is in use by another client (until **SyndCloseSynth** is called).

Units may be opened and closed multiple times during the lifetime of the DLL.

When this function is called, the driver should verify that the synthesiser exists. It should use information that is saved in the registry to find the location of the synth. It should not search for a synthesiser in this function.

When SAM starts up it will call **SyndOpenSynth** for each unit number that the driver reported in **SyndControl(SAMCONTROL_NUM_SYNTH)**. The driver should access the device, verify that it is working and leave it active or powered-up.

7.10 SyndCloseSynth

SyndCloseSynth is called to close access to the unit.

```
DWORD SyndCloseSynth (
    WORD unitid
    );
```

Parameters

unitid

The unit id of an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following.

SAMERROR_INVALID_UNIT – unitid is not valid.

SAMEROR_UNIT_CLOSED – the specified unit was not open.

Remarks

If the unit is battery driven or powered from the PC, the driver should (if possible) power-down the unit in this function (to conserve power on a portable system).

When SAM starts up, all units will be initially opened to verify their presence. Subsequently any units that have not been opened by the client after a period of time will be closed.

If all the units in a driver have been closed, the driver may be unloaded from memory.

7.11 SyndQuerySynth

This function gives additional information about each unit. SAM will call **SyndControl(SAMCONTROL_NUM_SYNTH)** to determine how many units there are. Unit information can be requested at any time, even when the unit is in use by another client, or the synthesiser is talking. Calls to **SyndQuerySynth** should not fail because of this.

```
DWORD SyndQuerySynth (
    DWORD        unitid,
    SYNTHPARAMS *lpunit,
    DWORD        langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range of 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

langid

This specifies the language that the synthesiser description string is reported in. English (44) must be available.

Return Value

This function fills the parameter block pointed to by lpunit with the unit's parameters.

The function returns zero if it succeeded.

If unitid is not valid (regardless of the validity of lpunit) the return value is **SAMERROR_INVALID_UNIT**.

SAMERROR_INVALID_ADDRESS if the address pointed to by param is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID will be returned if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

Remarks

See **SYNTHPARAMS** for a description of the parameters structure.

The driver should check that all parameters are valid and return the appropriate error code if they are not.

7.12 SyndQueryParam

Each synthesiser unit may have a variable number of parameters. The parameters for one synthesiser may not match any other synthesiser. **SyndQueryParam** allows the client to enumerate all the available parameters with descriptions. This interface allows additional parameters to be added in the future, and be compatible with older software

It is the driver's responsibility to report correct information about the unit. It can do this by either querying the unit directly or maintaining a data table of parameter information in memory.

SyndQueryParam may be called at any time, including whilst the unit is in use by another client.

```
DWORD SyndQueryParam (
    DWORD     unitid,  // unit number
    DWORD     pnum,    // parameter number
    SAMPARAM *param    // address of parameter
                       // info block
);
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

param

Address of a **SAMPARAM** structure to receive the data.

Return Value

The return value is zero if the function succeeded. Otherwise the possible error codes are:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_ADDRESS if the address pointed to by param is not valid or doesn't have write access.

Remarks

For each parameter the client wants to query, it will allocate the necessary memory for a **SAMPARAM** structure first.

See **SAMPARAM** for a description of a single parameter.

Client code example.

```
...
SAMPARAM param;
for (num=0;num<unitp.params;num++)
    {
    SyndQueryParam(unitid,num,&param);
    ...
    }
...
```

7.13 SyndQueryParamDesc

This function returns a description for a given parameter.

```
DWORD SyndQueryParamDesc (
    DWORD  unitid,          // unit id
    Long   pnum,            // parameter number
    DWORD  language,        // requested langid
    WCHAR *description,     // address of output
                            // buffer
    DWORD *size             // size of required
                            // buffer
);
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

language

The requested language of the description. This should be a value returned from **SyndGetLangId**.

description

This is a pointer to a buffer to receive the string. The string is in Unicode.

This value can be NULL if you want to know the length of the description string.

size

This is a pointer to a DWORD to receive the size of the required buffer in WORDS.

Return Value

If description is NULL then the number of WORDS in the resulting string (including the trailing zero) will be put in size. The client will use this value to allocate the necessary memory. If description is not NULL the requested string should be copied to the memory pointed to by description and the number of WORDS will be put in size.

If the function succeeds, the return code will be zero. Otherwise, the function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by description and size will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_ADDRESS – if the address pointed to by description is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if the language requested is not supported.

SAMERROR_NO_DESCRIPTION – if the driver does not have a description for this parameter in the specified language. This may occur if the driver has a partially complete set of descriptions for a particular language.

Remarks

No formatting should be provided. The description should be just the single name of the item, e.g. "Volume", "Voice" or "Speed". The string copied must be in Unicode.

The first letter should be capitalised.

The client may use this string to describe the parameter to the user and also to identify the parameter when loading and saving voice blocks.

The driver must support a full set of English descriptions. Any other languages are at the discretion of the driver.

size must always be filled in.

For parameters that have a valid id value, the meaning can be determined by the client and it may be able to describe it. For any additional parameters that don't have an official id value, it is recommended that the driver provide translations for all languages.

7.14 SyndQueryParamValue

This returns the numeric value for **SAMPARAM_COMPOUND** parameters. See the **SAMPARAM** structure for a description of this parameter type.

```
DWORD SyndQueryParamValue (
    DWORD  unitid,     // unit id
    long   pnum,       // parameter number
    DWORD  val,        // value index
    DWORD *pvalue      // pointer to DWORD to
                       // receive result
);
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

The parameter number.

val

This specifies the parameter value requested. val should be set to an integer from zero to range-1 where range is obtained from the SAMPARAM structure for this parameter.

pvalue

This is a pointer to a DWORD. If the function is successful, the result is placed here.

Return Value

If the function succeeded, the return value is zero. Otherwise, one of the following error codes may be returned.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number pnum is out of range.

SAMERROR_INVALID_ADDRESS – if the address pointed to by pvalue is not valid or doesn't have write access.

SAMERROR_INVALID_VAL – if val is beyond the range of valid values.

Remarks

This function is only relevant for parameters that are marked as **SAMPARAM_COMPOUND**. For other parameter types pvalue is the same as val. This function may be called with any valid pnum so for non-**SAMPARAM_COMPOUND** parameters the driver must return val.

7.15 SyndQueryParamChoice

This generates a string for a given parameter and value. The Client can call this function to get a description of each value for a given parameter. This is essential for Multi-choice parameters.

This function is also used to retrieve the description strings for other variables, such as voice names.

```
DWORD SyndQueryParamChoice (
    DWORD  unitid,          // unit id
    long   pnum,            // parameter number
    DWORD  val,             // actual parameter
                            // value
    DWORD  language,        // requested language
    WCHAR *description,     // address of buffer
    DWORD *size
    );
```

Parameters

unitid

The unit id of the synthesiser device.

pnum

If this is zero or greater then it indicates the requested parameter number that the client wants a description of.

If it is equal to -1 then you should supply voice names for default voices.

val

This is the actual value of the parameter. For numeric and multi-choice parameters, it must be in the range zero to range.

For compound parameters, it must be a valid value that has been retrieved with **SyndQueryParamValue**.

For voice names it is in the range from zero to the voice number.

language

The requested language of the description. This should be a value returned from **SyndGetLangId.**

description

This is a pointer to a buffer to receive the string. The string must be stored in Unicode.

If this value is NULL the client is requesting the length of the description string.

size

This is a pointer to a DWORD to receive the size of the required buffer in WORDS.

Return Value

If description is NULL then the number of WORDS in the resulting string (including the trailing zero) will be put in *size*. The client will use this value to allocate the necessary memory. If description is not NULL the requested string should be copied to the memory pointed to by description and the number of WORDS will be put in size.

If the function succeeds, the return code will be zero. Otherwise, the function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by description and size will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

SAMERROR_INVALID_VAL – if the val parameter is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by description is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if the language requested is not supported.

Remarks

No formatting should be provided. The description is just the single name of the item, e.g. "Male", "Female" or "Off". The string must be in Unicode.

The driver must support full descriptions for all parameters in English.

The driver must always fill in size.

If this function is called with a **SAMPARAM_NUMERIC** parameter the driver can create a string containing the simple numeric value of first+val and copy it into the description memory buffer pointed to by description. Alternatively, it could provide text descriptions for each possible value.

7.16 SyndGetLangId

This is used by the client to enumerate the languages available for descriptions of parameters. This is nothing to do with which languages the synthesiser can speak. The total number of available languages is stored in the **SYNTHPARAMS** structure. The client will call this function with language index number and retrieve a language identifier. The identifier should be used in subsequent calls to **SyndQuery...** functions.

```
DWORD SyndGetLangId (
    DWORD unitid,       // unit id
    DWORD index,        // n'th language
    DWORD *langid
);
```

Parameters

unitid

The unit id of the synthesiser device.

index

The language index number. This should be in the range from zero to langs-1, where langs is from the **SYNTHPARAMS** structure.

langid

This is a pointer to a DWORD to receive the language id code.

Return Value

If the function succeeded the return value is zero and the DWORD pointed to by langid will be filled with the actual language id for that language. The meaning of each language id is defined in ***sam.h***. These codes should be common between all drivers so that an application can identify each language.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

Remarks

The driver must support English. The code returned for each supported language should be defined in ***sam.h***.

7.17 SyndGetVoice

This function is used by the client to retrieve the default voice.

```
DWORD SyndGetVoice (
    DWORD  unitid,      // unit id
    Long   vnum,        // voice number
    DWORD *voiceblock  // address of voice block
                        // buffer
);
```

Parameters

unitid

The unit id of the synthesiser.

vnum

This value specifies the voice number that should be in the range from zero to the number of voices -1. A value of zero means the default voice.

voiceblock

This is a pointer to a memory buffer where the driver will store a copy of the voice block for the requested voice.

The size of the memory buffer required is `sizeof(DWORD)*synthparams.params`

The client will allocate the required memory before calling this function.

Return Value

If the function succeeds, the return value will be zero.

The function may return the following error codes instead. If an error occurs, the contents of the memory pointed to by voiceblock will not be changed.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_VNUM – if the voice number is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by voiceblock is not valid or doesn't have write access.

Remarks

Normally the client will enumerate all voices and cache the voice blocks internally.

The following parameters are required in a voice block, even if the range is one:

SAMID_LANGUAGE

8. SAM Braille Device Driver Programming Overview

This is a general overview of how the client uses the driver to access a Braille display. Normally the client will be SAM, but SAM is just passing the Braille requests directly from the access application or screen reader.

8.1 Driver loading

When the driver DLL has been loaded, SAM calls **BrldInitialise** once only to give the driver its registry key location so it can check its configuration.

8.2 Units

Each driver DLL may support zero or more units.

When the driver is loaded it should do any internal initialisation that doesn't involve a Braille display.

When **BrldControl(SAMCONTROL_NUM_BRAILLE)** is called the driver must return how many units have been configured (if any). SAM may then call **BrldOpenBraille** to access a unit.

It may be possible for the driver to be loaded without a Braille display plugged in or turned on. In this case, **BrldOpenBraille** should return an error code, so a screen reader may produce an audible message indicating that no display can be found and the user will be prompted to check their display.

If SAM is unable to locate a single working Braille display it will scan all drivers and call then with **BrldControl(SAMCONTROL_DETECT)** to attempt to locate a Braille display. In this way if the user moves or changes their Braille display their access software will be able to automatically use the new display.

In most cases, each driver will support only one Braille display.

If the driver is capable of detecting a display on one of several ports then it should do so during a call to **BrldControl(SAMCONTROL_DETECT)** providing this will not mess up any other hardware attached to the machine.

If a Braille display is not configured or cannot be detected at the correct location, the driver should return zero for a call to **BrldControl(SAMCONTROL_NUM_BRAILLE)**.

When the driver is unloaded, (in the call to **DLLMain**) it should shut down and clear any Braille display. Some Braille displays can be powered down at this point. Otherwise, the driver should power-down the Braille display in the call to **BrldCloseBraille**.

8.3 Opening and Closing

In order to validate access to the driver only one thread is allowed to open a device at one time. When a unit is opened it will not be opened again.

When a unit is closed using **BrldCloseBraille**, functions that produce Braille should fail.

Normally SAM will open each device as requested by each client. If two clients want to use the same device, SAM will provide the sharing interface and generate new handles for each client. The driver does not need to concern itself with this.

There is no need for unit handles because the SAM driver will only ever be called from SAM. **BrldOpenBraille** and **BrldCloseBraille** are only really useful to allow the driver to turn the device on and off. **BrldOpenBraille** also specifies the address of a callback function to return button push messages from the display.

8.4 Unit Capabilities

The client can query the capabilities of the Braille display device. It can do this before opening the device. API functions such as **BrldQueryBrailleEx**, **BrldQueryStrip** and **BrldGetLangId** may be called to get this information.

To find out information about each unit the client should call **BrldQueryBrailleEx**. This will fill a **BRAILLEPARAMS** structure with information. Including the number of strips the device supports, the client can then obtain information on each using **BrldQueryStrip** to return **STRIPPARAMS** structures.

The structures are of fixed size and format so the driver can simply copy a piece of static data when it is requested for it.

See **BRAILLEPARAMSEX** and **STRIPPARAMS** for a detailed description of the parameters.

8.5 Braille Parameters

Braille displays do not have parameters as speech synthesisers do. Information about a Braille display should be obtained by calling **BrldQueryBrailleEx**. This will fill in a **BRAILLEPARAMSEX** structure with all relevant information.

The client can then call **BrldQueryButtonStrip** to get a description for each button control on the display and **BrldIsKeyValidStrip** to ascertain whether specific button combinations are valid.

8.6 Parameter Descriptions

All description text will be stored in unicode. The client should call **BrldGetLangId** for the number of languages reported in the **BRAILLEPARAMSEX** structure to ascertain which languages the descriptions are in. UK English is guaranteed to be available.

8.7 Querying Button Functions

A client can ask the driver to provide suitable button combinations for certain actions which client software is likely to carry out. The client calls **BrldGetButtonCombinationStrip** and passes an action code from the actions list defined in ***brldacts.h***.

Client software should call this function to obtain sensible default button assignments for all the Braille functions it knows about. If no default button combination exists for a particular action, this function will return **SAMERROR_INVALID_VAL**.

8.8 Braille API Functions

Some of the Braille output functions pass data to the driver. This is passed by means of a pointer to a data block. While the thread of execution is in the driver the data in the block will remain valid. When the function has returned the client may delete, invalidate or overwrite the data areas. Therefore, the driver must make a copy of any relevant data inside the API call (if it will need to access it after the call has returned). The driver must never modify any of the data in buffers passed to it.

For functions that report back to the client, the driver must simply copy data to the address provided.

8.9 Text to Braille Output

The most vital part of the Braille driver is providing Braille output.

There are three main functions used for generating and controlling Braille output: **BrldClearDisplayStrip, BrldSetDisplayStrip** and **BrldSetCursorStrip**.

BrldClearDisplayStrip clears the cells on a strip or all the strips.

BrldSetDisplayStrip is used to set the dot pattern of a strip of cells. The data in a call to **BrldSetDisplayStrip** is of a fixed length as reported in the **STRIPPARAMS** structure. The data is in a language independent format, each bit representing a Braille dot.

BrldSetCursorStrip is used to set the position, shape and blink rate of a cursor on a strip.

8.10 Translation

The data to a call to **BrldSetDisplayStrip** is in a standard form, with each WORD of the data representing the Braille dot configuration of the cell.

In order to translate ASCII text into this standard form, a client can call **SamTranslateBraille**. This function performs a character-for-character translation.

Each WORD of the data is in the following form:

bit 0 - dot 1	bit 8 - blink dot 1
bit 1 - dot 4	bit 9 - blink dot 4
bit 2 - dot 2	bit 10 - blink dot 2
bit 3 - dot 5	bit 11 - blink dot 5
bit 4 - dot 3	bit 12 - blink dot 3
bit 5 - dot 6	bit 13 - blink dot 6
bit 6 - dot 7	bit 14 - blink dot 7
bit 7 - dot 8	bit 15 - blink dot 8

8.11 The callback function

When a client calls **BrldOpenBraille**, a callback function is specified. This function is used by the driver to pass button push information from the display to the client. The callback function will be called whenever a user pushes any button or combination of buttons on the display unit.

8.12 Errors

Hardware failure

If the hardware stops responding or is disconnected then any function may return **SAMERROR_UNIT_FAIL** to indicate to the client that the Braille display has stopped without prior warning. The driver should keep attempting to communicate with the display (without taking up too much CPU time!) until it starts to respond again, or **BrldCloseBraille** is called.

API Parameter errors

Generally, the driver should check all parameters to its functions and return appropriate error codes. In the case where there is potentially more than one applicable error code, the driver should try to check the parameters in the order they are listed in the function arguments. Otherwise, it can return any applicable error.

9. SAM Braille Device Driver Function Reference

This section provides a function reference for all the Braille display driver API functions. The driver DLL must support all functions listed here.

The structures are the same as defined in SAM.

9.1 BrldInitialise

This is called once only after the driver DLL has been loaded.

```
void BrldInitialise(
    CHAR *regkey,
    DWORD reserved
    );
```

Parameters

regkey

This is a pointer to a null terminated ANSI string that contains the path to the registry key for this driver. The driver should make a copy of this string.

reserved

This can be ignored and should be set to zero.

9.2 BrldCloseDown

This is called once only to close down the driver DLL. The driver should close all open units and release any memory it has allocated, close any open files and perform any other necessary clean up tasks in preparation for unloading.

```
void BrldCloseDown(void);
```

Remarks

There are no parameters and no return value.

9.3 BrldControl

This function is used for a number of functions, all to do with detecting and configuring one or more units.

The Braille display driver allows for multiple units. Each unit is a distinct Braille output device and is operated independently. Normally there will only be one unit unless the user has more than one Braille display installed.

```
DWORD BrldControl(DWORD mode);
```

Parameters

mode

This will be one of the following values and indicates what the function should do.

SAMCONTROL_NUM_BRAILLE

The function should return the number of Braille displays that it is configured for. It should not perform any detection or verification of devices. When the driver is manually configured or has auto-detected a display, it should save the physical location in the registry. It should return the number of displays last detected or configured.

If there are no units configured (which will be true the first time the driver is ever used) then this function should return zero.

Return value

The number of units configured should be returned.

SAMCONTROL_DETECT

When SAM is installed for the first time there will be no Braille displays configured. SAM will call this function in each driver to attempt to detect an initial Braille display.

The driver should perform any hardware detection it can to locate available displays. If the display is a plug 'n' play device then the Windows 95 device enumerator will have already found it and this driver can just query the registry as appropriate. If the display hardware has a proper Windows VxD driver that performs device detection, then again the registry can just be queried.

Otherwise, a Braille display should be searched for providing the search procedure is safe and does not break any other hardware. Providing the Win32 API is used to access hardware, this should not be a problem.

Regardless of how many Braille displays are found (if any) the driver should save any relevant information in the registry so that subsequent calls to **BrldControl(SAMCONTROL_NUM_BRAILLE)** will be accurate.

This function may also be called at the users request from the SAM control panel.

This function may be called whilst one or more units are open. In this case, the driver can assume that those units are present and correct and only search for Braille displays at other locations.

If the driver only supports one unit at a time then it should halt when it locates the first one.

Return value

BRLD_DETECT_OK should be returned if the detection sequence completed and the driver can verify the presence (or lack) of a Braille display.

BRLD_DETECT_CANT is the return code if the driver is unable to reliably detect a Braille display. SAM may subsequently call **BrldControl(SAMCONTROL_CONFIG)** later.

SAMCONTROL_CONFIG

This function gives the driver the opportunity to allow the user to enter any configuration information that the driver needs. SAM will call this function if the user selects CONFIG from the SAM control panel.

This function should display a dialog box with any relevant options in. The dialog box should contain the connection location of the Braille display. If the driver can auto detect then the box should contain 2 radio buttons for selecting between auto and manual configuration.

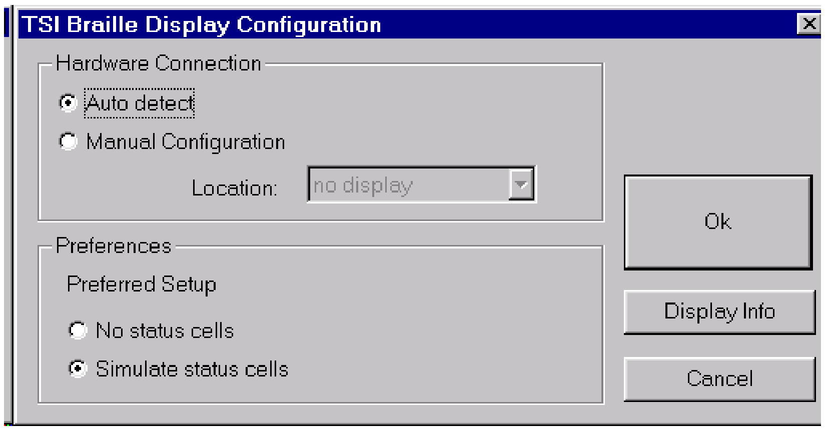
Any additional options are up to the driver. It could, for example, contain controls to select different models in the same display range (e.g. different display length) etc.

This function should not return to SAM until the dialog box is closed.

Any changes to the actual display configuration should not be made until the driver is closed and opened again (with a call to **BrldInitialise**) regardless of any changes made. The dialog box should contain two buttons, **Ok** and **Cancel**.

There may be a unit in use whilst this function is called (so that a screen reader can provide Braille access). When this function returns SAM may close all units and then call **BrldCloseDown** followed by **BrldInitialise**.

Example


TSI Braille Display Configuration

Hardware Connection

[✓ Auto detect]

Manual Configuration

Location: no display

Preferences

Preferred Setup

No status cells

[✓ Simulate status cells]

Return value

The return value tells SAM what to do next.

SAMCONFIG_CANCEL – if the user clicked on Cancel. SAM will do nothing.

SAMCONFIG_RELOAD – if a change has occurred that does not affect the location of the display, e.g. simulated status cells or device length. SAM will automatically close the driver with **BrldCloseDown** and re-open the driver with **BrldInitialise**. It will then notify applications of a driver configuration change.

SAMCONFIG_REDETECT – if a change affects the location (or existence) of a Braille display, e.g. com port location. SAM will then inform the user that changes will not take effect until the 'Detect All Devices Now' button is pressed. This will cause all drivers to be closed and re-started and the standard detection sequence to be initialised.

9.4 BrldQueryBrailleEx

This function gives information about each unit. SAM will call **BrldControl(SAMCONTROL_NUM_BRAILLE)** to determine how many units there are. Unit information can be requested at any time, even when the unit is in use by another client. Calls to **BrldQueryBraillle** should not fail because of this.

```
DWORD BrldQueryBraille(
    DWORD            unitid,
    BRAILLEPARAMSEX *lpunit,
    DWORD           *size,
    DWORD            langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range of 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

size

Points to a DWORD containing the size of the parameter block. SAM will initialise this to the size of the currently defined **BRAILLEPARAMSEX** structure. On exit the driver should update this with the size of data filled in. This allows the structure to be extended in future versions, without requiring drivers to be rewritten.

langid

This specifies the language that the Braille display description string is reported in. English (44) must be available.

Return Value

This function fills the parameter block pointed to by lpunit with the unit's parameters.

The function returns zero if it succeeded. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpunit)

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpunit is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

Remarks

See **BRAILLEPARAMSEX** for a description of the parameters structure.

9.5 BrldQueryStrip

This function gives information about each strip supported by the Braille unit. Strip information can be requested at any time without first opening the unit or while the Braille display is in use.

```
DWORD BrldQueryStrip(
    DWORD        unitid,
    DWORD        strip,
    STRIPPARAMS *lpstrip,
    DWORD       *size,
    DWORD        langid);
```

Parameters

unitid

This is the unit id of the device to query in the range 1 to the number of units.

strip

This is the index of the strip to query in the range 0 to the number of strips-1.

lpstrip

This is a pointer to the parameter block to be filled or NULL.

size

Points to a DWORD containing the size of the parameter block. SAM will initialise this to the size of the currently defined **STRIPPARAMS** structure. On exit the driver should update this with the size of data filled in. This allows the structure to be extended in future versions, without requiring drivers to be rewritten.

langid

This specifies the language that the strip description string is reported in. English (44) must be available.

Return Value

This function fills the parameter block pointed to by lpstrip with the strips parameters.

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpstrip).

SAMERROR_INVALID_STRIP – if strip is not valid.

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpstrip is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

9.6 BrldGetLangId

This is used by the client to enumerate the languages available for descriptions of parameters. The total number of available languages is stored in the **BRAILLEPARAMSEX** structure. The client will call this function with language index number and retrieve a language identifier. The identifier should be used in subsequent calls to **BrldQuery...** functions.

```
DWORD BrldGetLangId(
    DWORD  unitid,     // unit id
    DWORD  index,      // n'th language
    DWORD *langid
);
```

Parameters

unitid

The unit id of the device.

index

The language index number. This should be in the range from zero to langs-1, where langs is from the **BRAILLEPARAMSEX** structure.

langid

This is a pointer to a DWORD to receive the language id code.

Return Value

If the function succeeded the return value is zero and the DWORD pointed to by langid will be filled with the actual language id for that language. The meaning of each language id is defined in ***sam.h***. These codes should be common between all drivers so that an application can identify each language.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM – if the parameter number is out of range.

Remarks

The driver must support English. The code returned for each supported language should be defined in ***sam.h.***

9.7 BrldOpenBraille

BrldOpenBraille is used to open a Braille display unit. Only one process can open a unit at a time.

```
DWORD BrldOpenBraille(
    DWORD      unitid,
    BUTTONFUNC *fn,
    DWORD      devid
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive).

fn

The address of a callback function to handle button push messages from the unit.

devid

The identifier returned to the callback function.

Return Value

The function returns zero if it succeeded.

If the unit failed to open the return value is one of the following:

SAMERROR_INVALID_UNIT – the unitid value was invalid.

SAMERROR_UNIT_OPEN – the unit is already open.

SAMERROR_UNIT_FAIL – the physical device is not responding, turned off, unplugged or otherwise unusable.

Remarks

The client will call **BrldCloseBraille** to release ownership of the device.

Only one client is allowed to open each unit at one time. You must return SAMERROR_UNIT_OPEN for any subsequent attempts to open a unit that is in use by another client (until **BrldCloseBraille** is called).

Units may be opened and closed multiple times during the lifetime of the DLL.

When this function is called, the driver should verify that the Braille display exists. It should use information that is saved in the registry to find the location of the display. It should not search for a Braille display in this function.

When SAM starts up it will call **BrldOpenBraille** for each unit number that the driver reported in **BrldControl(SAMCONTROL_NUM_BRAILLE)**. The driver should access the device, verify that it is working and leave it active or powered-up.

9.8 BrldCloseBraille

BrldCloseBraille is called to close access to the unit.

```
DWORD BrldCloseBraille(
    DWORD unitid
    );
```

Parameters

unitid

The unit id of an open unit.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_UNIT - unitid is not valid.

SAMERROR_UNIT_CLOSED - the specified unit was not open.

Remarks

If the unit is battery driven or powered from the PC, the driver should (if possible) power-down the unit in this function (to conserve power on a portable system).

When SAM starts up, all units will be initially opened to verify their presence. Subsequently any units that have not been opened by the client after a period of time will be closed.

If all the units in a driver have been closed, the driver may be unloaded from memory.

9.9 BrldSetDisplayStrip

This is the main function used to send text to the driver.

```
DWORD BrldSetDisplay(
    DWORD  unitid,
    DWORD  strip,
    WORD  *dispdata
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

The strip number.

dispdata

This points to a block of words representing the dot pattern to be displayed on the cells of the strip. dispdata should contain the same number of WORDs as Braille cells as reported by **BrldQueryStrip**.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

SAMERROR_INVALID_ADDRESS – the address pointed to by dispdata is not valid or doesn't have read access.

Remarks

The format of each WORD in dispdata is in a standard form as follows:

bit 0 - dot 1	bit 8 - blink dot 1
bit 1 - dot 4	bit 9 - blink dot 4
bit 2 - dot 2	bit 10 - blink dot 2
bit 3 - dot 5	bit 11 - blink dot 5
bit 4 - dot 3	bit 12 - blink dot 3
bit 5 - dot 6	bit 13 - blink dot 6
bit 6 - dot 7	bit 14 - blink dot 7
bit 7 - dot 8	bit 15 - blink dot 8

If a dot is set to blink, the dot's normal state (i.e. bits 0-7 above) is periodically exclusive OR'd with the blink state (bits 8-15). This means that it will be possible to have blinking and inverse blinking dots. The distinction may not be clear to a Braille reader.

The data string must be of a fixed length as reported by calling **BrldQueryBrailleEx**. The first WORD represents the leftmost cell.

9.10 BrldClearDisplayStrip

This is used to clear the entire Braille display.

```
DWORD BrldClearDisplay(
    DWORD  unitid,
    DWORD  strip
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

strip

The strip number. If a value of **SAMSTRIP_ALL** is used, all strips should be cleared.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE - hBraille is not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

Remarks

All cells on the Braille affected braille strips will be blanked, and any cursor(s) will also be removed.

9.11 BrldSetCursorStrip

This is used to position, set the appearance and blink rate of a cursor.

```
DWORD BrldSetCursor(
    DWORD  unitid,
    DWORD  strip,
    long   pos,
    WORD   shape,
    DWORD  rate
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

strip

The strip number or **SAMSTRIP_ALL**.

pos

The position of the cursor in the range 0 to the number of cells - 1, 0 is the leftmost cell in the display area. A value of **SAMCURSOR_HIDE** should turn off the cursor on the particular strip, and if strip is **SAMSTRIP_ALL** all cursors on all strips should be turned off.

shape

The dot pattern of the cursor. When the cursor is "on", this pattern is OR'd with the dot pattern of the cell under the cursor.

rate

The blink rate of the cursor from 1 to the maxrate parameter as reported by **BrldQueryBrailleEx**. A value of 0 yields a static cursor. 1 is the slowest blink rate. The blink rate of the cursor will also affect any blinking text on the display.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_STRIP – strip is not valid.

SAMERROR_INVALID_VAL – pos or rate is out of range.

Remarks

The cursor will be repositioned and the display data unaffected.

The blink rate of the cursor will also affect any blinking text on the display.

See **BrldSetDisplayStrip** for the format of shape.

9.12 BrldQueryButtonStrip

This function is used to get a description of each button the Braille display has.

```
DWORD BrldQueryButton(
    DWORD  unitid,
    DWORD  strip,
    DWORD  num,
    WCHAR *description,
    DWORD *size,
    DWORD  langid
    );
```

Parameters

unitid

The number of a Braille display unit.

strip

The strip number.

num

For strips of type **SAMSTRIP_KEYS** this is the button index from 0 to the number of buttons-1, to get a description for. For all other strip types the call will return descriptions for the possible button combinations. num should range from zero to the number of button combinations-1 in a strip. The button combinations are returned by **BrldQueryStrip**.

description

The address where to store the description, or NULL.

size

The address to store the length of the description. Should be set to zero if no description is available.

langid

The language code of the description.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_STRIP – if the strip is not valid.

SAMERROR_INVALID_ADDRESS – if the address of description is invalid or does not have write access.

SAMERROR_INVALID_LANGID – if the language id is not valid.

Remarks

If description = NULL, this function returns the size of the description only.

Drivers do not have supply descriptions for all button combinations with strips of type **SAMSTRIP_BUTTONS,** for example when there are a large number of combinations for some of the subtypes such as dials / wheels / 2D. In which a size of zero should be returned.

9.13 BrldIsKeyValidStrip

This function determines if a particular key combination is valid for the unit.

```
BOOL BrldIsKeyValid(
    DWORD  unitid,
    DWORD  strip,
    BYTE  *pkey
    );
```

Parameters

unitid

The number of a Braille display unit.

strip

The strip number. This call if only valid for strips of type **SAMSTRIP_KEYS,** all other types should cause the function to return FALSE.

pKey

A pointer to a byte array forming a bit mask, indicating which buttons are pressed. Bit 0 corresponds to button zero. The size of the array should correspond to the number of buttons on the strip as returned by **BrlfQueryStrip**.

Return Value

This function returns TRUE if the unitid is valid and the button combination is valid.

Remarks

There is only one class of buttons that may be pressed in combination (except the action of shifting keys on routing buttons) and these will always be located on a strip of type **SAMSTRIP_KEYS**, however there may be certain combinations of buttons which are not valid (for example a combination of thumb keys and other buttons).

The order of buttons must correspond to the order given when descriptions are enumerated using the **BrldQueryButtonStrip** function.

9.14 BrldGetButtonCombinationStrip

This function is used to obtain sensible button default combinations for a variety of client software Braille related features.

```
DWORD BrldGetButtonCombination(
    DWORD  unitid,
    DWORD  action,
    DWORD *strip,
    BYTE  *pbutton
    );
```

Parameters

unitid

The unit number of a Braille unit.

action

One of the actions defined in ***brldacts.h***.

strip

This points to a DWORD to store the returned strip number for the button combination.

pbutton

This points to a buffer to store the returned recommended default button combination. If strip is of type **SAMSTRIP_KEYS** this will be a byte array forming a bit mask, indicating which button(s) are pressed. For other strip types, it will be to a **BRAILLEBUTTONPRESS** structure.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_ADDRESS – the address pointed to by pbutton is not valid or does not have write access.

SAMERROR_INVALID_VAL – there is no recommended default button combination.

SAMERROR_INVALID_UNIT – the unit number is not valid.

9.15 ButtonFuncStrip

This user-defined callback function is called by the driver to return any button push messages that a user generates by operating the buttons (of any type) on the Braille display. **ButtonFuncStrip** is just a name for the function.

```
void CALLBACK ButtonFuncStrip(
    DWORD  devid,
    DWORD  pbutton,
    DWORD  strip
    );
```

Parameters

devid

The devid of the device generating the message. This enables one callback function to handle multiple units. This parameter should be set to the value of devid passed to **BrldOpenBraille**.

pbutton

For strips of type **SAMSTRIP_KEYS** pbutton will point at a byte array of bits giving the mask of which keys have been pressed. The size of the array in bits is the same as number of buttons in the strip, rounded up to the nearest byte.

For all other strip types `pbutton` points to a **BRAILLEBUTTONPRESS** structure, containing the button number, combination with shifting keys and up/down information (if supported by the driver).

strip

The number of the strip on which the button resides.

Return Value

This function does not return a value.

Remarks

The function will copy the data pointed to by pbutton to a queue and notify the client via a Windows message. The driver is free to reuse this memory for another button pressed event immediately on exit from the function.

The following functions are from the SAM version 1 driver API. They may be used by SAM for backwards compatibility with existing drivers, but should not be implemented by any new drivers.

9.16 BrldQueryBraille

This function gives information about each unit. SAM will call **BrldControl(SAMCONTROL_NUM_BRAILLE)** to determine how many units there are. Unit information can be requested at any time, even when the unit is in use by another client. Calls to **BrldQueryBraillle** should not fail because of this.

```
DWORD BrldQueryBraille(
    DWORD          unitid,
    BRAILLEPARAMS *lpunit,
    DWORD          langid
    );
```

Parameters

unitid

This is the unit id of the device to query in the range of 1 to the number of units.

lpunit

This is a pointer to the parameter block to be filled or NULL.

langid

This specifies the language that the Braille display description string is reported in. English (44) must be available.

Return Value

This function fills the parameter block pointed to by lpunit with the unit's parameters.

The function returns zero if it succeeded. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_UNIT – if unitid is not valid (regardless of the validity of lpunit)

SAMERROR_INVALID_ADDRESS – if the address pointed to by lpunit is not valid or doesn't have write access.

SAMERROR_INVALID_LANGID – if langid is not a valid language. The language code for UK English is guaranteed to be supported in this function.

Remarks

See **BRAILLEPARAMS** for a description of the parameters structure.

The driver should check that all parameters are valid and return the appropriate error code if they are not.

9.17 BrldGetLangId

This is used by the client to enumerate the languages available for descriptions of parameters. The total number of available languages is stored in the **BRAILLEPARAMS** structure. The client will call this function with language index number and retrieve a language identifier. The identifier should be used in subsequent calls to **BrldQuery...** functions.

```
DWORD BrldGetLangId(
    DWORD  unitid,      // unit id
    DWORD  index,       // n'th language
    DWORD *langid
);
```

Parameters

unitid

The unit id of the device.

index

The language index number. This should be in the range from zero to langs-1, where langs is from the **BRAILLEPARAMS** structure.

langid

This is a pointer to a DWORD to receive the language id code.

Return Value

If the function succeeded the return value is zero and the DWORD pointed to by langid will be filled with the actual language id for that language. The meaning of each language id is defined in ***sam.h***. These codes should be common between all drivers so that an application can identify each language.

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_PNUM if the parameter number is out of range.

Remarks

The driver must support English. The code returned for each supported language should be defined in ***sam.h.***

9.18 BrldSetDisplay

This is the main function used to send text to the driver.

```
DWORD BrldSetDisplay(
    DWORD  unitid,      // unit identifier
    WORD  *dispdata,    // Dot pattern for main
                        // display area
    WORD  *statdata     // Dot pattern for
                        // status cells
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

dispdata

This points to a block of WORDs representing the dot pattern to be displayed in the main area of the unit. If dispdata = NULL, the display cells will remain in their current state. Dispdata should contain the same number of WORDs as Braille cells as reported by **BrldQueryBraille**.

statdata

As dispdata, this points to the dot pattern to be displayed on the status cells.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_ADDRESS – the address pointed to by dispdata or statdata is not valid or doesn't have read access.

Remarks

The format of each WORD in dispdata and statdata is in a standard form as follows:

bit 0 - dot 1	bit 8 - blink dot 1
bit 1 - dot 4	bit 9 - blink dot 4
bit 2 - dot 2	bit 10 - blink dot 2
bit 3 - dot 5	bit 11 - blink dot 5
bit 4 - dot 3	bit 12 - blink dot 3
bit 5 - dot 6	bit 13 - blink dot 6
bit 6 - dot 7	bit 14 - blink dot 7
bit 7 - dot 8	bit 15 - blink dot 8

If a dot is set to blink, the dot's normal state (i.e. bits 0-7 above) is periodically exclusive OR'd with the blink state (bits 8-15). This means that it will be possible to have blinking and inverse blinking dots. The distinction may not be clear to a Braille reader.

The data strings must be of a fixed length as reported by calling **BrldQueryBraille**.

The first WORD represents the leftmost cell. If displays have more than one line of cells, the length of the string should be width * height.

9.19 BrldClearDisplay

This is used to clear the entire Braille display.

```
DWORD BrldClearDisplay(
    DWORD unitid
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

Return Value

If the function is successful the return value is zero. Otherwise the return value may be one of the following:

SAMERROR_INVALID_HANDLE - hBraille is not valid.

Remarks

All cells on the Braille display will be blanked (both cells in the main area and status cells). Any cursor will also be removed.

This call is equivalent to a call to **BrldSetDisplay** and **BrldSetCursor**.

9.20 BrldSetCursor

This is used to position, set the appearance and blink rate of a cursor.

```
DWORD BrldSetCursor(
    DWORD  unitid,
    long   pos,
    WORD   shape,
    DWORD  rate
    );
```

Parameters

unitid

The unit number from 1 to the number of units (inclusive). This must be an open unit.

pos

The position of the cursor in the range 0 to the number of cells - 1, 0 is the leftmost cell in the display area. A value of -1 turns off the cursor.

shape

The dot pattern of the cursor. When the cursor is "on", this pattern is OR'd with the dot pattern of the cell under the cursor.

rate

The blink rate of the cursor from 1 to the maxrate parameter as reported by **BrldQueryBraille**. A value of 0 yields a static cursor. 1 is the slowest blink rate.

This blink rate also affects the rate that other characters blink at.

Return Value

If the function is successful, the return value is zero. Otherwise, the return value may be one of the following:

SAMERROR_INVALID_HANDLE – hBraille is not valid.

SAMERROR_INVALID_VAL – pos or rate is out of range.

Remarks

The cursor will be repositioned and the display data unaffected.

A cursor may not be positioned in the status area.

The blink rate of the cursor will also affect any blinking text on the display. Client software may choose to ignore this function completely and instead simulate the action of a cursor by using blinking text, or a static cursor shape.

9.21 BrldQueryButton

This function is used to get a description of each button the Braille display has.

```
DWORD BrldQueryButton(
    DWORD  unitid,
    DWORD  num,
    WCHAR *description,
    DWORD *size,
    DWORD  langid
    );
```

Parameters

unitid

The number of a Braille display unit.

num

The number from 0 to the number of buttons -1 to get a description for.

description

The address where to store the description, or NULL.

size

The address to store the length of the description.

langid

The language code of the description.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_UNIT – if the unitid is not valid.

SAMERROR_INVALID_ADDRESS – if the address of description is invalid or does not have write access.

SAMERROR_INVALID_LANGID – if the language id is not valid.

Remarks

If description = NULL, this function returns the size of the description only.

9.22 BrldIsKeyValid

This function determines if a particular key combination is valid for the unit.

```
BOOL BrldIsKeyValid(
    DWORD unitid,
    DWORD key
    );
```

Parameters

unitid

The number of a Braille display unit.

key

A bit mask indicating which of the keys is pressed. Bit 0 corresponds to button zero.

Return Value

This function returns TRUE if the unitid is valid and the button combination is valid.

Remarks

As there is only one class of buttons (apart from cursor routing buttons) there may be certain combinations of buttons which are not valid (for example a combination of thumb keys and other buttons).

The order of buttons must correspond to the order given when descriptions are enumerated using the **BrldQueryButton** function.

9.23 BrldGetButtonCombination

This function is used to obtain sensible button default combinations for a variety of client software Braille related features.

```
DWORD BrldGetButtonCombination(
     DWORD  unitid,     // Unit number
     DWORD  action,     // Braille related action
     DWORD *button      // Returned combination
     );
```

Parameters

unitid

The unit number of a Braille unit.

action

One of the actions defined in ***brldacts.h***.

button

This points to a DWORD to store the returned recommended default button combination.

Return Value

This function returns 0 on success or one of the following error codes:

SAMERROR_INVALID_ADDRESS – the address pointed to by button is not valid or does not have write access.

SAMERROR_INVALID_VAL – there is no recommended default button combination.

SAMERROR_INVALID_UNIT – the unit number is not valid.

9.24 ButtonFunc

This user-defined callback function is called by the driver to return any button push messages that a user generates by operating the buttons (of any type) on the Braille display. **ButtonFunc** is just a name for the function.

```
void CALLBACK ButtonFunc(
    DWORD devid,  // unit generating the button
    DWORD button, // button number
    DWORD type    // type of button
    );
```

Parameters

devid

The devid of the device generating the message. This enables one callback function to handle multiple units. This parameter should be set to the value of devid passed to **BrldOpenBraille**.

button

The number of the button pushed.

type

The type of the button pushed.

Return Value

This function does not return a value.

Remarks

The type parameter is used to specify what type of button the user pushed. Braille displays may have many different types of buttons as reported in the **BRAILLEPARAMS** structure. It will be one of:

SAMBRLBTN_NORMAL	Ordinary button
SAMBRLBTN_CSRDISP	Cursor routing button
SAMBRLBTN_CSRSTAT	Cursor route button over status cell
SAMBRLBTN_AUXDISP	Auxiliary routing button over display
SAMBRLBTN_AUXSTAT	Auxiliary routing button over status

If the button type is **SAMBRLBTN_NORMAL** then button represents a bit mask indicating which of the buttons were pressed. Bit 0 represents button 0. The order of buttons must correspond to the order given when descriptions are enumerated using the **BrldQueryButton** function.

Otherwise, button represents the number of the button.

10. SAM Device Driver Notification Functions

10.1 DrvrSetNotify

This is an optional function that can be provided by the driver. If present SAM will call it after the driver is loaded, supplying a pointer to a callback function. The driver can use this callback function to asynchronously notify SAM of particular events, such as reconfiguration.

```
void DrvrSetNotify(
    NOTIFYFUNC lpfn,        // Callback function
    DWORD      devid        // device identifier
    );
```

Parameters

lpfn

The call back function, see **NOTIFYFUNC**.

devid

The unique identifier of the driver, to be passed to used as the first parameter of callback.

10.2 NOTIFYFUNC

This callback function is to be used by the driver when it wishes to tell SAM its has detected its configuration has changed. Sam will then notify clients that is may be necessary to reopen devices and query their capabilities.

```
DWORD (CALLBACK *NOTIFYFUNC)(DWORD devid,
                             DWORD reason,
                             DWORD data);
```

Parameters

devid

The unique identifier of the driver from the **DrvrSetNotify** call.

reason

The type of notification, currently defined reason codes are:-

SAMCONFIG_RELOAD - driver settings have changed but the driver remains at the same location.

SAMCONFIG_REDETECT - driver location has changed requiring all devices to be re-detected by SAM. This may cause units to be renumbered and clients programs to ask users to reselect devices, so should be avoided if possible.

data

Additional data if required by the reason code, otherwise reserved and should be 0.

Return value

0 for success, otherwise an error code depending on reason.

Remarks

The actions by SAM as a result of reason codes **SAMCONFIG_RELOAD** and **SAMCONFIG_REDETECT** are similar to those when a driver configuration dialog is closed. See **SyndControl** and **BrldControl** for more information.

11. Change Log

Version	Changes
2.00 08/07/2002	Version 2.0 API extensions added **SamValidateVoice** (non implemented function) removed
2.01 29/08/2002	**STRIPPARAMS** cell and button handling simplified by having same number of routing buttons as cells in display/status/aux strips, any extra buttons going in their own strip. Added guidelines for driver writers, relating to strip arrangement.
2.02 01/10/2002	Device Driver Notification added. Notes on implementing both old and new driver APIs added.
2.03 07/11/2002	Strip types given more understandable names **SAMSTRIP_ROTARY** becomes **SAMSTRIP_DIAL** **SAMSTRIP_JOG** becomes **SAMSTRIP_WHEEL.** Situations where **SamQueryButtonStrip** may return an empty string clarified. **SAMWHEEL_TURND** and **SAMWHEEL_TURNU** added.
2.04 09/09/2003	**SAMCONTROL_SELECT_BRAILLE** added.
2.05 26/04/2005	Braille driver function parameter specifications corrected from HANDLE handle to DWORD unitid.