SANE Standard Version 1.04

2006-01-03

Contents

List of Figures

List of Tables

Chapter 1

Preface

The SANE standard is being developed by a group of free-software developers. The process is open to the public and comments as well as suggestions for improvements are welcome. Information on how to join the SANE development process can be found in Chapter 6.

The SANE standard is intended to streamline software development by providing a standard application programming interface to access raster scanner hardware. This should reduce the number of different driver implementations, thereby reducing the need for reimplementing similar code.

1.1 About This Document

This document is intended for developers who are creating either an application that requires access to raster scanner hardware and for developers who are implementing a SANE driver. It does not cover specific implementations of SANE components. Its sole purpose is to describe and define the SANE application interface that will enable any application on any platform to interoperate with any SANE backend for that platform.

The remainder of this document is organized as follows. Chapter 2 provides introductional material. Chapter 3 presents the environment SANE is designed for. Chapter 4 details the SANE Application Programmer Interface. Chapter 5 specifies the network protocol that can be used to implement the SANE API in a network transparent fashion. Finally, Chapter 6 gives information on how to join the SANE development process.

1.1.1 Typographic Conventions

Changes since the last revision of this document are highlighted like this:

Chapter 2

Introduction

SANE is an application programming interface (API) that provides standardized access to any raster image scanner hardware. The standardized interface allows to write just one driver for each scanner device instead of one driver for each scanner and application. The reduction in the number of required drivers provides significant savings in development time. More importantly, SANE raises the level at which applications can work. As such, it will enable applications that were previously unheard of in the UNIX world. While SANE is primarily targeted at a UNIX environment, the standard has been carefully designed to make it possible to implement the API on virtually any hardware or operating system.

SANE is an acronym for "Scanner Access Now Easy." Also, the hope is that SANE is sane in the sense that it will allow easy implementation of the API while accommodating all features required by today's scanner hardware and applications. Specifically, SANE should be broad enough to accommodate devices such as scanners, digital still and video cameras, as well as virtual devices like image file filters.

2.1 Terminology

An application that uses the SANE interface is called a SANE *frontend*. A driver that implements the SANE interface is called a SANE *backend*. A *meta backend* provides some means to manage one or more other backends.

Chapter 3

The SANE Environment

SANE is defined as a C-callable library interface. Accessing a raster scanner device typically consists of two phases: first, various controls of the scanner need to be setup or queried. In the second phase, one or more images are acquired.

Since the device controls are widely different from device to device, SANE provides a generic interface that makes it easy for a frontend to give a user access to all controls without having to understand each and every device control. The design principle used here is to abstract each device control into a SANE *option*. An option is a self-describing name/value pair. For example, the brightness control of a camera might be represented by an option called `brightness` whose value is an integer in the range from 0 to 255.

With self-describing options, a backend need not be concerned with *presentation* issues: the backend simply provides a list of options that describe all the controls available in the device. Similarly, there are benefits to the frontend: it need not be concerned with the *meaning* of each option. It simply provides means to present and alter the options defined by the backend.

3.1 Attaching to a SANE backend

The process through which a SANE frontend connects to a backend is platform dependent. Several possibilities exist:

- **Static linking:** A SANE backend may be linked directly into a frontend. While the simplest method of attaching to a backend, it is somewhat limited in functionality since the available devices is limited to the ones for which support has been linked in when the frontend was built. But even so static linking can be quite useful, particularly when combined with a backend that can access scanners via a network. Also, it is possible to support multiple backends simultaneously by implementing a meta backend that manages several backends that have been compiled in such a manner that they export unique function names. For example, a backend called `be` would normally export a function called `sane_read()`. If each backend would provide such a function, static linking would fail due to multiple conflicting definitions of the same symbol. This can be resolved by having backend `be` include a header file that has lines of the form:

```
#define sane_read be_sane_read
```

With definitions of this kind, backend `be` will export function name `be_sane_read()`. Thus, all backends will export unique names. As long as a meta backend knows about these names, it is possible to combine several backends at link time and select and use them dynamically at runtime.

- **Dynamic linking:** A simpler yet more powerful way to support multiple backends is to exploit dynamic linking on platforms that support it. In this case, a frontend is linked against a shared library that implements any SANE backend. Since each dynamically linked backend exports the same set of global symbols (all starting with the prefix `sane_`), the dynamic library that gets loaded at runtime does not necessarily have to be the same one as one the frontend got linked against. In other words, it is possible to switch the backend by installing the appropriate backend dynamic library.

  More importantly, dynamic linking makes it easy to implement a meta backend that loads other backends *on demand*. This is a powerful mechanism since it allows adding new backends merely by installing a shared library and updating a configuration file.

- **Network connection:** Arguably the ultimate way to attach to a scanner is by using the network to connect to a backend on a remote machine. This makes it possible to scan images from any host in the universe, as long as there is a network connection to that host and provided the user is permitted to access that scanner.

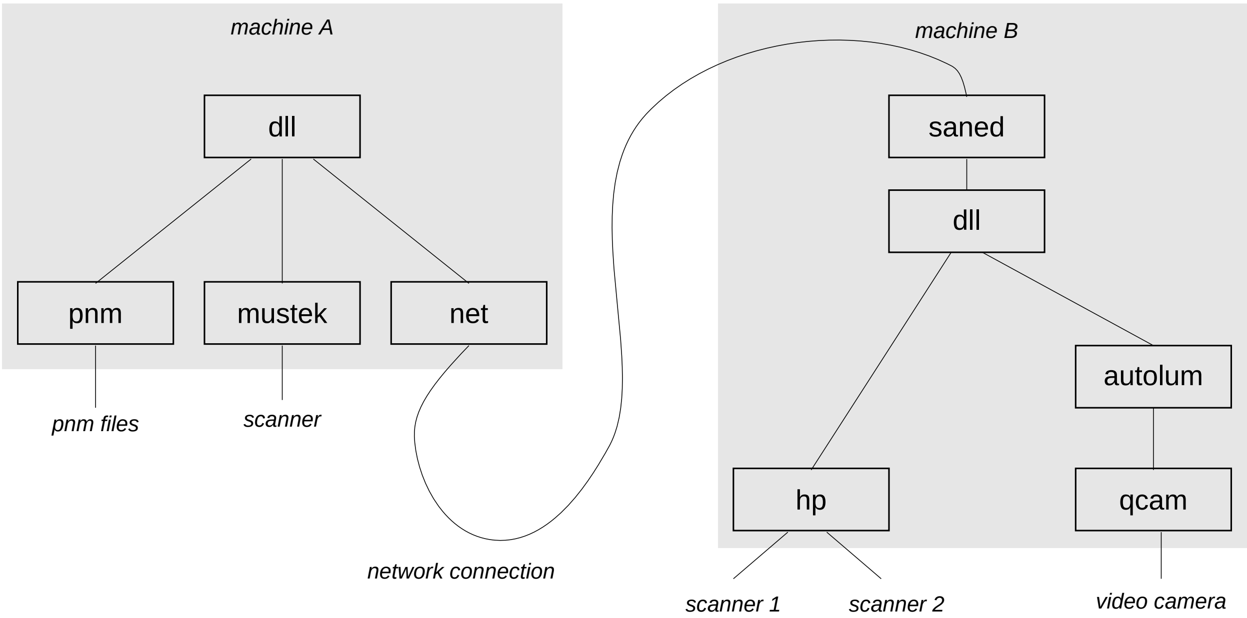

Figure 3.1: Example SANE Hiearchy

The above discussion lists just a few ways for frontends to attach to a backend. It is of course possible to combine these solutions to provide an entire hierarchy of SANE backends. Such a hierarchy is depicted in Figure 3.1. The figure shows that machine A uses a dynamic-linking based meta backend called `dll` to access the backends called `pnm`, `mustek`, and `net`. The first two are real backends, whereas the last one is a meta backend that provides network transparent access to remote scanners. In the figure, machine B provides non-local access to its scanners through the SANE frontend called `saned`. The `saned` in turn has access to the `hp` and `autolum` backends through another instance of the `dll` backend. The `autolum` meta backend is used to automatically adjust the luminance (brightness) of the image data acquired by the camera backend called `qcam`.

Note that a meta backend really is both a frontend and a backend at the same time. It is a frontend from the viewpoint of the backends that it manages and a backend from the viewpoint of the frontends that access it. The name "meta backend" was chosen primarily because the SANE standard describes the interface from the viewpoint of a (real) frontend.

3.2 Image Data Format

Arguably the most important aspect of an image acquisition system is how images are represented. The SANE approach is to define a simple yet powerful representation that is sufficient for vast majority of applications and devices. While the representation is simple, the interface has been defined carefully to allow extending it in the future without breaking backwards compatibility. Thus, it will be possible to accommodate future applications or devices that were not anticipated at the time this standard was created.

A SANE image is a rectangular area. The rectangular area is subdivided into a number of rows and columns. At the intersection of each row and column is a quadratic pixel. A pixel consists of one or more sample values. Each sample value represents one channel (e.g., the red channel). Each sample value has a certain bit depth. The bit depth is fixed for the entire image and can be as small as one bit. Valid bit depths are 1, 8, or 16 bits per sample. If a device's natural bit depth is something else, it is up to the driver to scale the sample values appropriately (e.g., a 4 bit sample could be scaled by a factor of four to represent a sample value of depth 8).

3.2.1 Image Transmission

The SANE API transmits an image as a sequence of frames. Each frame covers the same rectangular area as the entire image, but may contain only a subset of the channels in the final image. For example, a red/green/blue image could either be transmitted as a single frame that contains the sample values for all three channels or it could be transmitted as a sequence of three frames: the first frame containing the red channel, the second the green channel, and the third the blue channel.

Conceptually, each frame is transmitted a byte at a time. Each byte may contain 8 sample values (for an image bit depth of 1), one full sample value (for an image bit depth of 8), or a partial sample value (for an image bit depth of 16 or bigger). In the latter case, the bytes of each sample value are transmitted in the machine's native byte order. For depth 1, the leftmost pixel is stored in the most significant bit, and the rightmost pixel in the least significant bit.

Backend Implementation Note

> A network-based meta backend will have to ensure that the byte order in image data is adjusted appropriately if necessary. For example, when the meta backend attaches to the server proxy,

the proxy may inform the backend of the server's byte order. The backend can then apply the adjustment if necessary. In essence, this implements a "receiver-makes-right" approach.

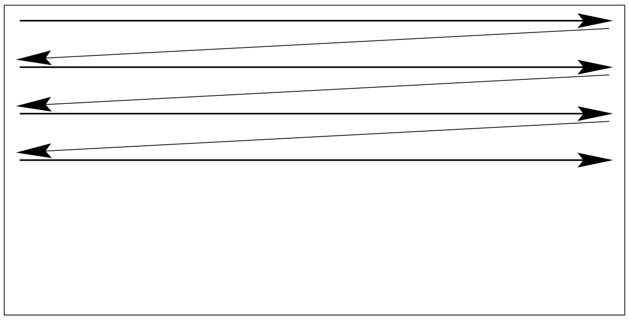

Figure 3.2: Transfer order of image data bytes

The order in which the sample values in a frame are transmitted is illustrated in Figure 3.2. As can be seen, the values are transmitted row by row and each row is transmitted from left-most to right-most column. The left-to-right, top-to-bottom transmission order applies when the image is viewed in its normal orientation (as it would be displayed on a screen, for example).

If a frame contains multiple channels, then the channels are transmitted in an interleaved fashion. Figure 3.3 illustrates this for the case where a frame contains a complete red/green/blue image with a bit-depth of 8. For a bit depth of 1, each byte contains 8 sample values of a *single* channel. In other words, a bit depth 1 frame is transmitted in a byte interleaved fashion.

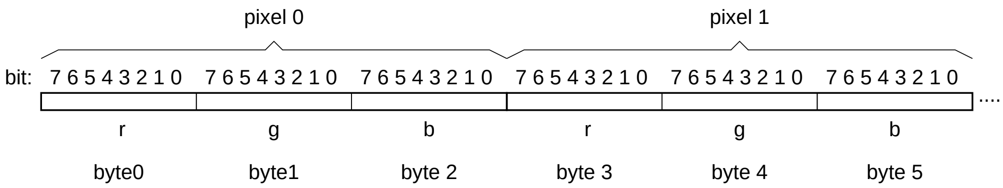

Figure 3.3: Bit and byte order or image data

When transmitting an image frame by frame, the frontend needs to know what part of the image a frame represents (and how many frames it should expect). For that purpose, the SANE API tags every frame with a type. This version of the SANE standard supports the following frame types:

`SANE_FRAME_GRAY`**:** The frame contains a single channel of data that represents sample values from a spectral band that covers the human visual range. The image consists of this frame only.

`SANE_FRAME_RGB`**:** The frame contains three channels of data that represent sample values from the red, green, and blue spectral bands. The sample values are interleaved in the order red, green, and blue. The image consists of this frame only.

SANE_FRAME_RED: The frame contains one channel of data that represents sample values from the red spectral band. The complete image consists of three frames: SANE_FRAME_RED, SANE_FRAME_GREEN, and SANE_FRAME_BLUE. The order in which the frames are transmitted chosen by the backend.

SANE_FRAME_GREEN: The frame contains one channel of data that represents sample values from the green spectral band. The complete image consists of three frames: SANE_FRAME_RED, SANE_FRAME_GREEN, and SANE_FRAME_BLUE. The order in which the frames are transmitted chosen by the backend.

SANE_FRAME_BLUE: The frame contains one channel of data that represents sample values from the blue spectral band. The complete image consists of three frames: SANE_FRAME_RED, SANE_FRAME_GREEN, and SANE_FRAME_BLUE. The order in which the frames are transmitted chosen by the backend.

In frames of type SANE_FRAME_GRAY, when the bit depth is 1 there are only two sample values possible, 1 represents minimum intensity (black) and 0 represents maximum intensity (white). For all other bit depth and frame type combinations, a sample value of 0 represents minimum intensity and larger values represent increasing intensity.

The combination of bit depth 1 and SANE_FRAME_RGB (or SANE_FRAME_RED, SANE_FRAME_GREEN, SANE_FRAME_BLUE) is rarely used and may not be supported by every frontend.

Chapter 4

The SANE Application Programmer Interface (API)

This Section defines version 1 of the SANE application programmer interface (API). Any SANE frontend must depend on the interface defined in this section only. Converseley, any SANE backend must implement its functionality in accordance with this specification. The interface as documented here is declared as a C callable interface in a file called `sane/sane.h`. This file should normally be included via a C pre-processor directive of the form:

```
#include <sane/sane.h>
```

4.1 Version Control

The SANE standard is expected to evolve over time. Whenever a change to the SANE standard is made that may render an existing frontend or backend incompatible with the new standard, the major version number must be increased. Thus, any frontend/backend pair is compatible provided the major version number of the SANE standard they implement is the same. A frontend may implement backwards compatiblity by allowing major numbers that are smaller than the expected major number (provided the frontend really can cope with the older version). In contrast, a backend always provides support for one and only one version of the standard. If a specific application does require that two different versions of the same backend are accessible at the same time, it is possible to do so by installing the two versions under different names.

SANE version control also includes a minor version number and a build revision. While control of these numbers remains with the implementor of a backend, the recommended use is as follows. The minor version is incremented with each official release of a backend. The build revision is increased with each build of a backend.

The SANE API provides the following five macros to manage version numbers.

`SANE_CURRENT_MAJOR`**:** The value of this macro is the number of the SANE standard that the interface implements.

`SANE_VERSION_CODE(`*maj*`,`*min*`,`*bld*`)`: This macro can be used to build a monotonically increasing version code. A SANE version code consists of the SANE standard major version number (*maj*), the minor version number *min*, and the build revision of a backend (*bld*). The major and minor version numbers must be in the range 0...255 and the build revision must be in the range 0...65535.

Version codes are monotonic in the sense that it is possible to apply relational operators (e.g., equality or less-than test) directly on the version code rather than individually on the three components of the version code.

Note that the major version number alone determines whether a frontend/backend pair is compatible. The minor version and the build revision are used for informational and bug-fixing purposes only.

`SANE_VERSION_MAJOR(`*vc*`)`: This macro returns the major version number component of the version code passed in argument *vc*.

`SANE_VERSION_MINOR(`*vc*`)`: This macro returns the minor version number component of the version code passed in argument *vc*.

`SANE_VERSION_BUILD(`*vc*`)`: This macro returns the build revision component of the version code passed in argument *vc*.

4.2 Data Types

4.2.1 Base Types

The SANE standard is based on just two SANE-specific base types: the SANE byte and word.

```
typedef some-scalar-type SANE_Byte;
typedef some-scalar-type SANE_Word;
```

`SANE_Byte` must correspond to some scalar C type that is capable of holding values in the range 0 to 255. `SANE_Word` must be capable of holding any of the following:

- the truth values `SANE_FALSE` and `SANE_TRUE`
- signed integers in the range $-2^{31} \ldots 2^{31} - 1$
- fixed point values in the range $-32768 \ldots 32767.9999$ with a resolution of $1/65536$
- 32 bits (for bit sets)

Note that the SANE standard does not define what C type `SANE_Byte` and `SANE_Word` map to. For example, on some platforms, the latter may map to `long int` whereas on others it may map to `int`. A portable SANE frontend or backend must therefore not depend on a particular mapping.

4.2.2 Boolean Type

SANE_Bool is used for variables that can take one of the two truth values SANE_FALSE and SANE_TRUE. The former value is defined to be 0, whereas the latter is 1.[1] The C declarations for this type are given below.

```
#define SANE_FALSE      0
#define SANE_TRUE       1
typedef SANE_Word SANE_Bool;
```

Note that SANE_Bool is simply an alias of SANE_Word. It is therefore always legal to use the latter type in place of the former. However, for clarity, it is recommended to use SANE_Bool whenever a given variable or formal argument has a fixed interpretation as a boolean object.

4.2.3 Integer Type

SANE_Int is used for variables that can take integer values in the range -2^{32} to $2^{31} - 1$. Its C declaration is given below.

```
typedef SANE_Word SANE_Int;
```

Note that SANE_Int is simply an alias of SANE_Word. It is therefore always legal to use the latter type in place of the former. However, for clarity, it is recommended to use SANE_Int whenever a given variable or formal argument has a fixed interpretation as an integer object.

4.2.4 Fixed-point Type

SANE_Fixed is used for variables that can take fixed point values in the range -32768 to 32767.9999 with a resolution of $1/65535$. The C declarations relating to this type are given below.

```
#define SANE_FIXED_SCALE_SHIFT  16
typedef SANE_Word SANE_Fixed;
```

The macro SANE_FIXED_SCALE_SHIFT gives the location of the fixed binary point. This standard defines that value to be 16, which yields a resolution of $1/65536$.

Note that SANE_Fixed is simply an alias of SANE_Word. It is therefore always legal to use the latter type in place of the former. However, for clarity, it is recommended to use SANE_Fixed whenever a given variable or formal argument has a fixed interpretation as a fixed-point object.

For convenience, SANE also defines two macros that convert fixed-point values to and from C double floating point values.

[1] This is different from ANSI C where any non-zero integer value represents logical TRUE.

`SANE_FIX(`*d*`)`: Returns the largest SANE fixed-point value that is smaller than the double value *d*. No range checking is performed. If the value of *d* is out of range, the result is undefined.

`SANE_UNFIX(`*w*`)`: Returns the nearest double machine number that corresponds to fixed-point value *w*.

SANE does *not* require that the following two expressions hold true (even if the values of *w* and *d* are in range):

```
SANE_UNFIX(SANE_FIX(d)) == d
SANE_FIX(SANE_UNFIX(w)) == w
```

In other words, conversion between fixed and double values may be lossy. It is therefore recommended to avoid repeated conversions between the two representations.

4.2.5 Text

Character Type

Type `SANE_Char` represents a single text character or symbol. At present, this type maps directly to the underlying C `char` type (typically one byte). The encoding for such characters is currently fixed as ISO LATIN-1. Future versions of this standard may map this type to a wider type and allow multi-byte encodings to support internationalization. As a result of this, care should be taken to avoid the assumption that `sizeof(SANE_Char)==sizeof(char)`.

```
typedef char SANE_Char;
```

String Type

Type `SANE_String` represents a text string as a sequence of C `char` values. The end of the sequence is indicated by a `'\0'` (NUL) character.

```
typedef SANE_Char *SANE_String;
typedef const SANE_Char *SANE_String_Const;
```

The type `SANE_String_Const` is provided by SANE to enable declaring strings whose contents is unchangable. Note that in ANSI C, the declaration

```
const SANE_String str;
```

declares a string pointer that is constant (not a string pointer that points to a constant value).

4.2.6 Scanner Handle Type

Access to a scanner is provided through an opaque type called `SANE_Handle`. The C declaration of this type is given below.

```
typedef void *SANE_Handle;
```

While this type is declared to be a void pointer, an application must not attempt to interpret the value of a `SANE_Handle`. In particular, SANE does not require that a value of this type is a legal pointer value.

4.2.7 Status Type

Most SANE operations return a value of type `SANE_Status` to indicate whether the completion status of the operation. If an operation completes successfully, `SANE_STATUS_GOOD` is returned. In case of an error, a value is returned that indicates the nature of the problem. The complete list of available status codes is listed in Table 4.1. It is recommended to use function `sane_strstatus()` to convert status codes into a legible string.

Symbol	Code	Description
SANE_STATUS_GOOD	0	Operation completed succesfully.
SANE_STATUS_UNSUPPORTED	1	Operation is not supported.
SANE_STATUS_CANCELLED	2	Operation was cancelled.
SANE_STATUS_DEVICE_BUSY	3	Device is busy—retry later.
SANE_STATUS_INVAL	4	Data or argument is invalid.
SANE_STATUS_EOF	5	No more data available (end-of-file).
SANE_STATUS_JAMMED	6	Document feeder jammed.
SANE_STATUS_NO_DOCS	7	Document feeder out of documents.
SANE_STATUS_COVER_OPEN	8	Scanner cover is open.
SANE_STATUS_IO_ERROR	9	Error during device I/O.
SANE_STATUS_NO_MEM	10	Out of memory.
SANE_STATUS_ACCESS_DENIED	11	Access to resource has been denied.

Table 4.1: Status Codes

4.2.8 Device Descriptor Type

Each SANE device is represented by a structure of type `SANE_Device`. The C declaration of this type is given below.

```
typedef struct
  {
    SANE_String_Const name;
```

```
    SANE_String_Const vendor;
    SANE_String_Const model;
    SANE_String_Const type;
  }
SANE_Device;
```

The structure provides the unique name of the scanner in member `name`. It is this unique name that should be passed in a call to `sane_open()`. The format of this name is completely up to the backend. The only constraints are that the name is unique among all devices supported by the backend and that the name is a legal SANE text string. To simplify presentation of unique names, their length should not be excessive. It is *recommended* that backends keep unique names below 32 characters in length. However, applications *must* be able to cope with arbitrary length unique names.

The remaining members in the device structure provide additional information on the device corresponding to the unique name. Specifically, members `vendor`, `model`, and `type` are single-line strings that give information on the vendor (manufacturer), model, and the type of the device. For consistency's sake, the following strings should be used when appropriate (the lists will be expanded as need arises):

Vendor Strings	
AGFA	Microtek
Abaton	Minolta
Acer	Mitsubishi
Apple	Mustek
Artec	NEC
Avision	Nikon
CANON	Plustek
Connectix	Polaroid
Epson	Relisys
Fujitsu	Ricoh
Hewlett-Packard	Sharp
IBM	Siemens
Kodak	Tamarack
Lexmark	UMAX
Logitech	Noname

Type Strings
film scanner
flatbed scanner
frame grabber
handheld scanner
multi-function peripheral
sheetfed scanner
still camera
video camera
virtual device

Table 4.2: Predefined Device Information Strings

Note that vendor string `Noname` can be used for virtual devices that have no physical vendor associated. Also, there are no predefined model name strings since those are vendor specific and therefore completely under control of the respective backends.

4.2.9 Option Descriptor Type

Option descriptors are at the same time the most intricate and powerful type in the SANE standard. Options are used to control virtually all aspects of device operation. Much of the power of the SANE API stems from the fact that most device controls are completely described by their respective option descriptor. Thus, a frontend can control a scanner abstractly, without requiring knowledge as to what the purpose of any given option is. Conversely, a scanner can describe its controls without requiring knowledge of how the frontend operates. The C declaration of the `SANE_Option_Descriptor` type is given below.

```
typedef struct
  {
    SANE_String_Const name;
    SANE_String_Const title;
    SANE_String_Const desc;
    SANE_Value_Type type;
    SANE_Unit unit;
    SANE_Int size;
    SANE_Int cap;
    SANE_Constraint_Type constraint_type;
    union
      {
        const SANE_String_Const *string_list;
        const SANE_Word *word_list;
        const SANE_Range *range;
      }
    constraint;
  }
SANE_Option_Descriptor;
```

Option Name

Member `name` is a string that uniquely identifies the option. The name must be unique for a given device (i.e., the option names across different backends or devices need not be unique). The option name must consist of lower-case ASCII letters (`a`–`z`), digits (`0`–`9`), or the dash character (`-`) only. The first character must be a lower-case ASCII character (i.e., not a digit or a dash).

Option Title

Member `title` is a single-line string that can be used by the frontend as a title string. This should typically be a short (one or two-word) string that is chosen based on the function of the option.

Option Description

Member `desc` is a (potentially very) long string that can be used as a help text to describe the option. It is the responsibility of the frontend to break the string into managable-length lines. Newline characters in this string should be interpreted as paragraph breaks.

Option Value Type

Member `type` specifies the type of the option value. The possible values for type `SANE_Value_Type` are described in Table 4.3.

Symbol	Code	Description
`SANE_TYPE_BOOL`	0	Option value is of type `SANE_Bool`.
`SANE_TYPE_INT`	1	Option value is of type `SANE_Int`.
`SANE_TYPE_FIXED`	2	Option value is of type `SANE_Fixed`.
`SANE_TYPE_STRING`	3	Option value is of type `SANE_String`.
`SANE_TYPE_BUTTON`	4	An option of this type has no value. Instead, setting an option of this type has an option-specific side-effect. For example, a button-typed option could be used by a backend to provide a means to select default values or to the tell an automatic document feeder to advance to the next sheet of paper.
`SANE_TYPE_GROUP`	5	An option of this type has no value. This type is used to group logically related options. A group option is in effect up to the point where another group option is encountered (or up to the end of the option list, if there are no other group options). For group options, only members `title` and `type` are valid in the option descriptor.

Table 4.3: Option Value Types (`SANE_Value_Type`)

Option Value Unit

Member `unit` specifies what the physical unit of the option value is. The possible values for type `SANE_Unit` are described in Table 4.4. Note that the specified unit is what the SANE backend expects. It is entirely up to a frontend as to how these units a presented to the user. For example, SANE expresses all lengths in millimeters. A frontend is generally expected to provide appropriate conversion routines so that a user can express quantities in a customary unit (e.g., inches or centimeters).

Symbol	Code	Description
`SANE_UNIT_NONE`	0	Value is unit-less (e.g., page count).
`SANE_UNIT_PIXEL`	1	Value is in number of pixels.
`SANE_UNIT_BIT`	2	Value is in number of bits.
`SANE_UNIT_MM`	3	Value is in millimeters.
`SANE_UNIT_DPI`	4	Value is a resolution in dots/inch.
`SANE_UNIT_PERCENT`	5	Value is a percentage.
`SANE_UNIT_MICROSECOND`	6	Value is time in μ-seconds.

Table 4.4: Physical Units (`SANE_Unit`)

Option Value Size

Member size specifies the size of the option value (in bytes). This member has a slightly different interpretation depending on the type of the option value:

SANE_TYPE_STRING: The size is the maximum size of the string. For the purpose of string size calcuations, the terminating NUL character is considered to be part of the string. Note that the terminating NUL character must always be present in string option values.

SANE_TYPE_INT, SANE_TYPE_FIXED: The size must be a positive integer multiple of the size of a SANE_Word. The option value is a vector of length

$$\texttt{size}/\texttt{sizeof}(\texttt{SANE\_Word}).$$

SANE_TYPE_BOOL: The size must be set to sizeof(SANE_Word).

SANE_TYPE_BUTTON, SANE_TYPE_GROUP: The option size is ignored.

Option Capabilities

Member cap describes what capabilities the option posseses. This is a bitset that is formed as the inclusive logical OR of the capabilities described in Table 4.5. The SANE API provides the following to macros to test certain features of a given capability bitset:

SANE_OPTION_IS_ACTIVE(*cap*): This macro returns SANE_TRUE if and only if the option with the capability set *cap* is currently active.

SANE_OPTION_IS_SETTABLE(*cap*): This macro returns SANE_TRUE if and only if the option with the capability set *cap* is software settable.

Option Value Constraints

It is often useful to constrain the values that an option can take. For example, constraints can be used by a frontend to determine how to represent a given option. Member constraint_type indicates what constraint is in effect for the option. The constrained values that are allowed for the option are described by one of the union members of member constraint. The possible values of type SANE_Constraint_Type and the interpretation of the constraint union is described in Table 4.6.

4.3 Operations

4.3.1 sane_init

This function must be called before any other SANE function can be called. The behavior of a SANE backend is undefined if this function is not called first or if the status code returned by sane_init is

Symbol	Code	Description
SANE_CAP_SOFT_SELECT	1	The option value can be set by a call to sane_control_option().
SANE_CAP_HARD_SELECT	2	The option value can be set by user-intervention (e.g., by flipping a switch). The user-interface should prompt the user to execute the appropriate action to set such an option. This capability is mutually exclusive with SANE_CAP_SOFT_SELECT (either one of them can be set, but not both simultaneously).
SANE_CAP_SOFT_DETECT	4	The option value can be detected by software. If SANE_CAP_SOFT_SELECT is set, this capability *must* be set. If SANE_CAP_HARD_SELECT is set, this capability may or may not be set. If this capability is set but neither SANE_CAP_SOFT_SELECT nor SANE_CAP_HARD_SELECT are, then there is no way to control the option. That is, the option provides read-out of the current value only.
SANE_CAP_EMULATED	8	If set, this capability indicates that an option is not directly supported by the device and is instead emulated in the backend. A sophisticated frontend may elect to use its own (presumably better) emulation in lieu of an emulated option.
SANE_CAP_AUTOMATIC	16	If set, this capability indicates that the backend (or the device) is capable to picking a reasonable option value automatically. For such options, it is possible to select automatic operation by calling sane_control_option() with an action value of SANE_ACTION_SET_AUTO.
SANE_CAP_INACTIVE	32	If set, this capability indicates that the option is not currently active (e.g., because it's meaningful only if another option is set to some other value).
SANE_CAP_ADVANCED	64	If set, this capability indicates that the option should be considered an "advanced user option." A frontend typically displays such options in a less conspicuous way than regular options (e.g., a command line interface may list such options last or a graphical interface may make them available in a seperate "advanced settings" dialog).

Table 4.5: Option Capabilities

Symbol	Code	Description
`SANE_CONSTRAINT_NONE`	0	The value is unconstrained. The option can take any of the values possible for the option's type.

`SANE_CONSTRAINT_RANGE`	1	This constraint is applicable to integer and fixed-point valued options only. It constrains the option value to a possibly quantized range of numbers. Option descriptor member `constraint.range` points to a range of the type `SANE_Range`. This type is illustrated below:
		
		`typedef struct`
		`  {`
		`    SANE_Word min;`
		`    SANE_Word max;`
		`    SANE_Word quant;`
		`  }`
		`SANE_Range;`
		
		All three members in this structure are interpreted according to the option value type (`SANE_TYPE_INT` or `SANE_TYPE_FIXED`). Members `min` and `max` specify the minimum and maximum values, respectively. If member `quant` is non-zero, it specifies the quantization value. If l is the minimum value, u the maximum value and q the (non-zero) quantization of a range, then the legal values are $v = k \cdot q + l$ for all non-negative integer values of k such that $v <= u$.

`SANE_CONSTRAINT_WORD_LIST`	2	This constraint is applicable to integer and fixed-point valued options only. It constrains the option value to a list of numeric values. Option descriptor member `constraint.word_list` points to a list of words that enumerates the legal values. The first element in that list is an integer (`SANE_Int`) that specifies the length of the list (not counting the length itself). The remaining elements in the list are interpreted according to the type of the option value (`SANE_TYPE_INT` or `SANE_TYPE_FIXED`).
`SANE_CONSTRAINT_STRING_LIST`	3	This constraint is applicable to string-valued options only. It constrains the option value to a list of strings. The option descriptor member `constraint.string_list` points to a `NULL` terminated list of strings that enumerate the legal values for the option value.

Table 4.6: Option Value Constraints

different from SANE_STATUS_GOOD. The version code of the backend is returned in the value pointed to by version_code. If that pointer is NULL, no version code is returned. Argument authorize is either a pointer to a function that is invoked when the backend requires authentication for a specific resource or NULL if the frontend does not support authentication.

```
SANE_Status sane_init (SANE_Int * version_code,
                       SANE_Authorization_Callback authorize);
```

The authorization function may be called by a backend in response to any of the following calls:

sane_open, sane_control_option, sane_start

If a backend was initialized without authorization function, then authorization requests that cannot be handled by the backend itself will fail automatically and the user may be prevented from accessing protected resources. Backends are encouraged to implement means of authentication that do not require user assistance. E.g., on a multi-user system that authenticates users through a login process a backend could automatically lookup the apporpriate password based on resource- and user-name.

The authentication function type has the following declaration:

```
#define SANE_MAX_USERNAME_LEN   128
#define SANE_MAX_PASSWORD_LEN   128

typedef void (*SANE_Authorization_Callback)
    (SANE_String_Const resource,
     SANE_Char username[SANE_MAX_USERNAME_LEN],
     SANE_Char password[SANE_MAX_PASSWORD_LEN]);
```

Three arguments are passed to the authorization function: resource is a string specifying the name of the resource that requires authorization. A frontend should use this string to build a user-prompt requesting a username and a password. The username and password arguments are (pointers to) an array of SANE_MAX_USERNAME_LEN and SANE_MAX_PASSWORD_LEN characters, respectively. The authorization call should place the entered username and password in these arrays. The returned strings *must* be ASCII-NUL terminated.

4.3.2 sane_exit

This function must be called to terminate use of a backend. The function will first close all device handles that still might be open (it is recommended to close device handles explicitly through a call to sane_close(), but backends are required to release all resources upon a call to this function). After this function returns, no function other than sane_init() may be called (regardless of the status value returned by sane_exit(). Neglecting to call this function may result in some resources not being released properly.

```
void sane_exit (void);
```

4.3.3 `sane_get_devices`

This function can be used to query the list of devices that are available. If the function executes successfully, it stores a pointer to a `NULL` terminated array of pointers to `SANE_Device` structures in `*device_list`. The returned list is guaranteed to remain unchanged and valid until (a) another call to this function is performed or (b) a call to `sane_exit()` is performed. This function can be called repeatedly to detect when new devices become available. If argument `local_only` is true, only local devices are returned (devices directly attached to the machine that SANE is running on). If it is false, the device list includes all remote devices that are accessible to the SANE library.

```
SANE_Status sane_get_devices (const SANE_Device *** device_list,
                              SANE_Bool local_only);
```

This function may fail with `SANE_STATUS_NO_MEM` if an insufficient amount of memory is available.

Backend Implementation Note

> SANE does not require that this function is called before a `sane_open()` call is performed. A device name may be specified explicitly by a user which would make it unnecessary and undesirable to call this function first.

4.3.4 `sane_open`

This function is used to establish a connection to a particular device. The name of the device to be opened is passed in argument `name`. If the call completes successfully, a handle for the device is returned in `*h`. As a special case, specifying a zero-length string as the device requests opening the first available device (if there is such a device).

```
SANE_Status sane_open (SANE_String_Const name, SANE_Handle * h);
```

This function may fail with one of the following status codes.

- `SANE_STATUS_DEVICE_BUSY`**:** The device is currently busy (in use by somebody else).
- `SANE_STATUS_INVAL`**:** The device name is not valid.
- `SANE_STATUS_IO_ERROR`**:** An error occured while communicating with the device.
- `SANE_STATUS_NO_MEM`**:** An insufficent amount of memory is available.
- `SANE_STATUS_ACCESS_DENIED`**:** Access to the device has been denied due to insufficient or invalid authentication.

4.3.5 `sane_close`

This function terminates the association between the device handle passed in argument `h` and the device it represents. If the device is presently active, a call to `sane_cancel()` is performed first. After this function returns, handle `h` must not be used anymore.

```
void sane_close (SANE_Handle h);
```

4.3.6 sane_get_option_descriptor

This function is used to access option descriptors. The function returns the option descriptor for option number n of the device represented by handle h. Option number 0 is guaranteed to be a valid option. Its value is an integer that specifies the number of options that are available for device handle h (the count includes option 0). If n is not a valid option index, the function returns NULL. The returned option descriptor is guaranteed to remain valid (and at the returned address) until the device is closed.

```
const SANE_Option_Descriptor *
    sane_get_option_descriptor (SANE_Handle h, SANE_Int n);
```

4.3.7 sane_control_option

This function is used to set or inquire the current value of option number n of the device represented by handle h. The manner in which the option is controlled is specified by parameter a. The possible values of this parameter are described in more detail below. The value of the option is passed through argument v. It is a pointer to the memory that holds the option value. The memory area pointed to by v must be big enough to hold the entire option value (determined by member size in the corresponding option descriptor). The only exception to this rule is that when setting the value of a string option, the string pointed to by argument v may be shorter since the backend will stop reading the option value upon encountering the first NUL terminator in the string. If argument i is not NULL, the value of *i will be set to provide details on how well the request has been met. The meaning of this argument is described in more detail below.

```
SANE_Status sane_control_option (SANE_Handle h, SANE_Int n,
                                 SANE_Action a, void *v,
                                 SANE_Int * i);
```

The way the option is affected by a call to this function is controlled by parameter a which is a value of type SANE_Action. The possible values and their meaning is described in Table 4.7.

Symbol	Code	Description
SANE_ACTION_GET_VALUE	0	Get current option value.
SANE_ACTION_SET_VALUE	1	Set option value. The option value passed through argument v may be modified by the backend if the value cannot be set exactly.
SANE_ACTION_SET_AUTO	2	Turn on automatic mode. Backend or device will automatically select an appropriate value. This mode remains effective until overridden by an explicit set value request. The value of parameter v is completely ignored in this case and may be NULL.

Table 4.7: Action Values (SANE_Action)

After setting a value via an action value of SANE_ACTION_SET_VALUE, additional information on how well the request has been met is returned in *i (if i is non-NULL). The returned value is a bitset that may contain any combination of the values described in Table 4.8.

Symbol	Code	Description
SANE_INFO_INEXACT	1	This value is returned when setting an option value resulted in a value being selected that does not exactly match the requested value. For example, if a scanner can adjust the resolution in increments of 30dpi only, setting the resolution to 307dpi may result in an actual setting of 300dpi. When this happens, the bitset returned in *i has this member set. In addition, the option value is modified to reflect the actual (rounded) value that was used by the backend. Note that inexact values are admissible for strings as well. A backend may choose to "round" a string to the closest matching legal string for a constrained string value.
SANE_INFO_RELOAD_OPTIONS	2	The setting of an option may affect the value or availability of one or more *other* options. When this happens, the SANE backend sets this member in *i to indicate that the application should reload all options. This member may be set if and only if at least one option changed.
SANE_INFO_RELOAD_PARAMS	4	The setting of an option may affect the parameter values (see sane_get_parameters()). If setting an option affects the parameter values, this member will be set in *i. Note that this member may be set even if the parameters did not actually change. However, it is guaranteed that the parameters never change without this member being set.

Table 4.8: Additional Information Returned When Setting an Option

This function may fail with one of the following status codes.

- SANE_STATUS_UNSUPPORTED**:** The operation is not supported for the specified handle and option number.
- SANE_STATUS_INVAL**:** The option value is not valid.
- SANE_STATUS_IO_ERROR**:** An error occured while communicating with the device.
- SANE_STATUS_NO_MEM**:** An insufficent amount of memory is available.
- SANE_STATUS_ACCESS_DENIED**:** Access to the option has been denied due to insufficient or invalid authentication.

4.3.8 sane_get_parameters

This function is used to obtain the current scan parameters. The returned parameters are guaranteed to be accurate between the time a scan has been started (sane_start() has been called) and the completion of that request. Outside of that window, the returned values are best-effort estimates of what the parameters will be when sane_start() gets invoked. Calling this function before a scan has actually started allows, for example, to get an estimate of how big the scanned image will be. The parameters passed to this function are the handle h of the device for which the parameters should be obtained and a pointer p to a parameter structure. The parameter structure is described in more detail below.

```
SANE_Status sane_get_parameters (SANE_Handle h,
                                 SANE_Parameters * p);
```

The scan parameters are returned in a structure of type SANE_Parameters. The C declaration of this structure is given below.

```
typedef struct
  {
    SANE_Frame format;
    SANE_Bool last_frame;
    SANE_Int lines;
    SANE_Int depth;
    SANE_Int pixels_per_line;
    SANE_Int bytes_per_line;
  }
SANE_Parameters;
```

Member format specifies the format of the next frame to be returned. The possible values for type SANE_Frame are described in Table 4.9. The meaning of these values is described in more detail in Section 3.2.

Symbol	Code	Description
SANE_FRAME_GRAY	0	Band covering human visual range.
SANE_FRAME_RGB	1	Pixel-interleaved red/green/blue bands.
SANE_FRAME_RED	2	Red band of a red/green/blue image.
SANE_FRAME_GREEN	3	Green band of a red/green/blue image.
SANE_FRAME_BLUE	4	Blue band of a red/green/blue image.

Table 4.9: Frame Format (SANE_Frame)

Member last_frame is set to SANE_TRUE if and only if the frame that is currently being acquired (or the frame that will be acquired next if there is no current frame) is the last frame of a multi frame image (e.g., the current frame is the blue component of a red, green, blue image).

Member `lines` specifies how many scan lines the frame is comprised of. If this value is -1, the number of lines is not known a priori and the frontend should call `sane_read()` until it returns a status of `SANE_STATUS_EOF`.

Member `bytes_per_line` specifies the number of bytes that comprise one scan line.

Member `depth` specifies the number of bits per sample.

Member `pixels_per_line` specifies the number of pixels that comprise one scan line.

Assume B is the number of channels in the frame, then the bit depth d (as given by member `depth`) and the number of pixels per line n (as given by this member `pixels_per_line`) are related to c, the number of bytes per line (as given by member `bytes_per_line`) as follows:

$$c >= \begin{cases} B \cdot \lfloor (n+7)/8 \rfloor & \text{if } d = 1 \\ B \cdot n \cdot d/8 & \text{if } d > 1 \end{cases}$$

Note that the number of bytes per line can be larger than the minimum value imposed by the right side of this equation. A frontend must be able to properly cope with such "padded" image formats.

4.3.9 `sane_start`

This function initiates aquisition of an image from the device represented by handle `h`.

```
SANE_Status sane_start (SANE_Handle h);
```

This function may fail with one of the following status codes.

- `SANE_STATUS_CANCELLED`**:** The operation was cancelled through a call to `sane_cancel`.
- `SANE_STATUS_DEVICE_BUSY`**:** The device is busy. The operation should be retried later.
- `SANE_STATUS_JAMMED`**:** The document feeder is jammed.
- `SANE_STATUS_NO_DOCS`**:** The document feeder is out of documents.
- `SANE_STATUS_COVER_OPEN`**:** The scanner cover is open.
- `SANE_STATUS_IO_ERROR`**:** An error occurred while communicating with the device.
- `SANE_STATUS_NO_MEM`**:** An insufficent amount of memory is available.
- `SANE_STATUS_INVAL`**:** The scan cannot be started with the current set of options. The frontend should reload the option descriptors, as if `SANE_INFO_RELOAD_OPTIONS` had been returned from a call to `sane_control_option()`, since the device's capabilities may have changed.

4.3.10 sane_read

This function is used to read image data from the device represented by handle h. Argument buf is a pointer to a memory area that is at least maxlen bytes long. The number of bytes returned is stored in *len. A backend must set this to zero when a status other than SANE_STATUS_GOOD is returned. When the call succeeds, the number of bytes returned can be anywhere in the range from 0 to maxlen bytes.

```
SANE_Status sane_read (SANE_Handle h, SANE_Byte * buf,
                       SANE_Int maxlen, SANE_Int * len);
```

If this function is called when no data is available, one of two things may happen, depending on the I/O mode that is in effect for handle h.

1. If the device is in blocking I/O mode (the default mode), the call blocks until at least one data byte is available (or until some error occurs).
2. If the device is in non-blocking I/O mode, the call returns immediately with status SANE_STATUS_GOOD and with *len set to zero.

The I/O mode of handle h can be set via a call to sane_set_io_mode().

This function may fail with one of the following status codes.

SANE_STATUS_CANCELLED**:** The operation was cancelled through a call to sane_cancel.

SANE_STATUS_EOF**:** No more data is available for the current frame.

SANE_STATUS_JAMMED**:** The document feeder is jammed.

SANE_STATUS_NO_DOCS**:** The document feeder is out of documents.

SANE_STATUS_COVER_OPEN**:** The scanner cover is open.

SANE_STATUS_IO_ERROR**:** An error occurred while communicating with the device.

SANE_STATUS_NO_MEM**:** An insufficent amount of memory is available.

SANE_STATUS_ACCESS_DENIED**:** Access to the device has been denied due to insufficient or invalid authentication.

4.3.11 sane_cancel

This function is used to immediately or as quickly as possible cancel the currently pending operation of the device represented by handle h.

```
void sane_cancel (SANE_Handle h);
```

This function can be called at any time (as long as handle h is a valid handle) but usually affects long-running operations only (such as image is acquisition). It is safe to call this function asynchronously (e.g., from within a signal handler). It is important to note that completion of this operaton does *not* imply that the currently pending operation has been cancelled. It only guarantees that cancellation has been *initiated*. Cancellation completes only when the cancelled call returns (typically with a status value of SANE_STATUS_CANCELLED). Since the SANE API does not require any other operations to be re-entrant, this implies that a frontend must *not* call any other operation until the cancelled operation has returned.

4.3.12 sane_set_io_mode

This function is used to set the I/O mode of handle h. The I/O mode can be either blocking or non-blocking. If argument m is SANE_TRUE, the mode is set to non-blocking mode, otherwise it's set to blocking mode. This function can be called only after a call to sane_start() has been performed.

```
SANE_Status sane_set_io_mode (SANE_Handle h, SANE_Bool m);
```

By default, newly opened handles operate in blocking mode. A backend may elect not to support non-blocking I/O mode. In such a case the status value SANE_STATUS_UNSUPPORTED is returned. Blocking I/O must be supported by all backends, so calling this function with argument m set to SANE_FALSE is guaranteed to complete successfully.

This function may fail with one of the following status codes:

SANE_STATUS_INVAL: No image acquisition is pending.

SANE_STATUS_UNSUPPORTED: The backend does not support the requested I/O mode.

4.3.13 sane_get_select_fd

This function is used to obtain a (platform-specific) file-descriptor for handle h that is readable if and only if image data is available (i.e., when a call to sane_read() will return at least one byte of data). If the call completes successfully, the select file-descriptor is returned in *fd.

```
SANE_Status sane_get_select_fd (SANE_Handle h, SANE_Int *fd);
```

This function can be called only after a call to sane_start() has been performed and the returned file-descriptor is guaranteed to remain valid for the duration of the current image acquisition (i.e., until sane_cancel() or sane_start() get called again or until sane_read() returns with status SANE_STATUS_EOF). Indeed, a backend must guarantee to close the returned select file descriptor at the point when the next sane_read() call would return SANE_STATUS_EOF. This is necessary to ensure the application can detect when this condition occurs without actually having to call sane_read().

A backend may elect not to support this operation. In such a case, the function returns with status code SANE_STATUS_UNSUPPORTED.

Note that the only operation supported by the returned file-descriptor is a host operating-system dependent test whether the file-descriptor is readable (e.g., this test can be implemented using select() or poll() under UNIX). If any other operation is performed on the file descriptor, the behavior of the backend becomes unpredictable. Once the file-descriptor signals "readable" status, it will remain in that state until a call to sane_read() is performed. Since many input devices are very slow, support for this operation is strongly encouraged as it permits an application to do other work while image acquisition is in progress.

This function may fail with one of the following status codes:

SANE_STATUS_INVAL: No image acquisition is pending.

SANE_STATUS_UNSUPPORTED: The backend does not support this operation.

4.3.14 `sane_strstatus`

This function can be used to translate a SANE status code into a printable string. The returned string is a single line of text that forms a complete sentence, but without the trailing period (full-stop). The function is guaranteed to never return `NULL`. The returned pointer is valid at least until the next call to this function is performed.

```
const SANE_String_Const sane_strstatus (SANE_Status status);
```

4.4 Code Flow

The code flow for the SANE API is illustrated in Figure 4.1. Functions `sane_init()` and `sane_exit()` initialize and exit the backend, respectively. All other calls must be performed after initialization and before exiting the backend.

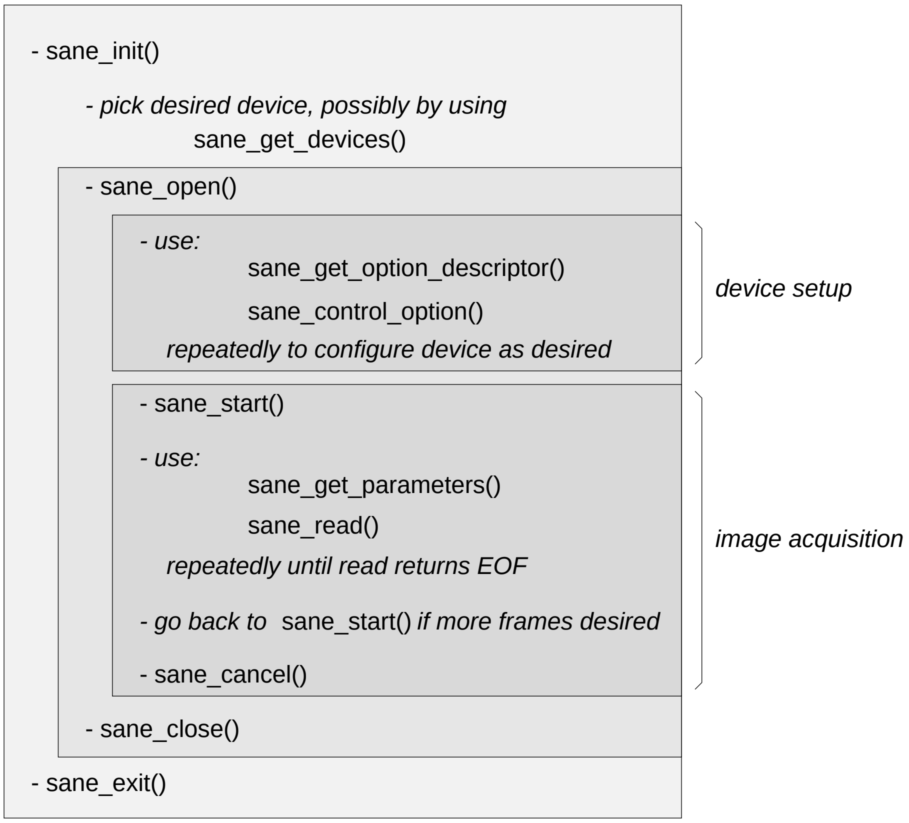

Figure 4.1: Code flow

Function `sane_get_devices()` can be called any time after `sane_init()` has been called. It returns the list of the devices that are known at the time of the call. This list may change over time since some devices may be turned on or off or a remote host may boot or shutdown between different calls. It should be noted that this operation may be relatively slow since it requires contacting all configured devices (some of which may be on remote hosts). A frontend may therefore want to provide the ability for a user to directly select a desired device without requiring a call to this function.

Once a device has been chosen, it is opened using a call to `sane_open()`. Multiple devices can be open at any given time. A SANE backend must not impose artificial constraints on how many devices can be open at any given time.

An opened device can be setup through the corresponding device handle using functions `sane_get_option_descriptor()` and `sane_control_option()`. While setting up a device, obtaining option descriptors and setting and reading of option values can be mixed freely. It is typical for a frontend to read out all available options at the beginning and then build a dialog (either graphical or a command-line oriented option list) that allows to control the available options. It should be noted that the number of options is fixed for a given handle. However, as options are set, other options may become active or inactive. Thus, after setting an option, it maybe necessary to re-read some or all option descriptors. While setting up the device, it is also admissible to call `sane_get_parameters()` to get an estimate of what the image parameters will look like once image acquisition begins.

The device handle can be put in blocking or non-blocking mode by a call to `sane_set_io_mode()`. Devices are required to support blocking mode (which is the default mode), but support for non-blocking I/O is strongly encouraged for operating systems such as UNIX.

After the device is setup properly, image acquisition can be started by a call to `sane_start()`. The backend calculates the exact image parameters at this point. So future calls to `sane_get_parameters()` will return the exact values, rather than estimates. Whether the physical image acquisition starts at this point or during the first call to `sane_read()` is unspecified by the SANE API. If non-blocking I/O and/or a select-style interface is desired, the frontend may attempt to call `sane_set_io_mode()` and/or `sane_get_select_fd()` at this point. Either of these functions may fail if the backend does not support the requested operation.

Image data is collected by repeatedly calling `sane_read()`. Eventually, this function will return an end-of-file status (`SANE_STATUS_EOF`). This indicates the end of the current frame. If the frontend expects additional frames (e.g., the individual channels in of a red/green/blue image or multiple images), it can call `sane_start()` again. Once all desired frames have been acquired, function `sane_cancel()` must be called. This operation can also be called at any other time to cancel a pending operation. Note that `sane_cancel()` must be called even if the last read operation returned `SANE_STATUS_EOF`.

When done using the device, the handle should be closed by a call to `sane_close()`. Finally, before exiting the application, function `sane_exit()` must be called. It is important not to forget to call this function since otherwise some resources (e.g., temporary files or locks) may remain unclaimed.

4.5 Well-Known Options

While most backend options are completely self-describing, there are a cases where a user interface might want to special-case the handling of certain options. For example, the scan area is typically defined by four options that specify the top-left and bottom-right corners of the area. With a graphical user interface, it would be tedious to force the user to type in these four numbers. Instead, most such interfaces will want to present to the user a preview (low-resolution scan) of the scanner surface and let the user pick the scan area by dragging a rectangle into the desired position. For this reason, the SANE API specifies a small number of option names that have well-defined meanings.

4.5.1 Option Number Count

Option number 0 has an empty string as its name. The value of this option is of type `SANE_TYPE_INT` and it specifies the total number of options available for a given device (the count includes option number 0). This means that there are two ways of counting the number of options available: a frontend can either cycle through all option numbers starting at one until `sane_get_option_descriptor()` returns `NULL`, or a frontend can directly read out the value of option number 0.

4.5.2 Scan Resolution Option

Option `resolution` is used to select the resolution at which an image should be acquired. The type of this option is either `SANE_TYPE_INT` or `SANE_TYPE_FIXED`. The unit is `SANE_UNIT_DPI` (dots/inch).

This option is not mandatory, but if a backend does support it, it must implement it in a manner consistent with the above definition.

4.5.3 Preview Mode Option

The boolean option `preview` is used by a frontend to inform the backend when image acquisition should be optimized for speed, rather than quality ("preview mode"). When set to `SANE_TRUE`, preview mode is in effect, when set to `SANE_FALSE` image acquisition should proceed in normal quality mode. The setting of this option *must not* affect any other option. That is, as far as the other options are concerned, the preview mode is completely side effect free. A backend can assume that the frontend will take care of appropriately setting the scan resolution for preview mode (through option `resolution`). A backend is free to override the `resolution` value with its own choice for preview mode, but it is advised to leave this choice to the frontend wherever possible.

This option is not mandatory, but if a backend does support it, it must implement it in a manner consistent with the above definition.

4.5.4 Scan Area Options

The four most important well-known options are the ones that define the scan area. The scan area is defined by two points (x/y coordinate pairs) that specify the top-left and the bottom-right corners. This is illustrated in Figure 4.2. Note that the origin of the coordinate system is at the top-left corner of the scan surface as seen by the sensor (which typically is a mirror image of the scan surface seen by the user). For this reason, the top-left corner is the corner for which the abscissa and ordinate values are simultaneously the *smallest* and the bottom-right corner is the corner for which the abscissa and ordinate values are simulatenously the

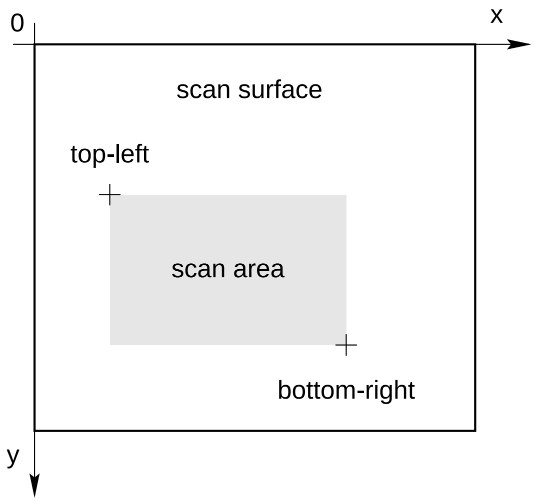

Figure 4.2: Scan area options

largest. If this coordinate system is not natural for a given device, it is the job of the backend to perform the necessary conversions.

The names of the four options that define the scan area are given in the table below:

Name	**Description**
`tl-x`	Top-left x coordinate value
`tl-y`	Top-left y coordinate value
`br-x`	Bottom-right x coordinate value
`br-y`	Bottom-right y coordinate value

There are several rules that should be followed by front and backends regarding these options:

- Backends must attach a unit of either pixels (`SANE_UNIT_PIXEL`) or millimeters (`SANE_UNIT_MM`) to these options. The unit of all four options must be identical.
- Whenever meaningful, a backend should attach a range or a word-list constraint to these options.
- A frontend can determine the size of the scan surface by first checking that the options have range constraints associated. If a range or word-list constraints exist, the frontend can take the minimum and maximum values of one of the x and y option range-constraints to determine the scan surface size.
- A frontend must work properly with any or all of these options missing.

Chapter 5

Network Protocol

The SANE interface has been designed to facilitate network access to image acquisition devices. In particular, most SANE implementations are expected to support a network backend (net client) and a corresponding network daemon (net server) that allows accessing image acquisition devices through a network connection. Network access is useful in several situations:

- To provide controlled access to resources that are inaccessible to a regular user. For example, a user may want to access a device on a host where she has no account on. With the network protocol, it is possible to allow certain users to access scanners without giving them full access to the system.

  Controlling access through the network daemon can be useful even in the local case: for example, certain backends may require root privileges to access a device. Rather than installing each frontend as setuid-root, a system administrator could instead install the SANE network daemon as setuid-root. This enables regular users to access the privileged device through the SANE daemon (which, presumably, supports a more fine-grained access control mechanism than the simple setuid approach). This has the added benefit that the system administrator only needs to trust the SANE daemon, not each and every frontend that may need access to the privileged device.

- Network access provides a sense of ubiquity of the available image acquisition devices. For example, in a local area network environment, this allows a user to log onto any machine and have convenient access to any resource available to any machine on the network (subject to permission constraints).

- For devices that do not require physical access when used (e.g., video cameras), network access allows a user to control and use these devices without being in physical proximity. Indeed, if such devices are connected to the Internet, access from any place in the world is possible.

The network protocol described in this chapter has been design with the following goals in mind:

1. Image transmission should be efficient (have low encoding overhead).
2. Accessing option descriptors on the client side must be efficient (since this is a very common operation).

3. Other operations, such as setting or inquiring the value of an option are less performance critical since they typically require explicit user action.
4. The network protocol should be simple and easy to implement on any host architecture and any programming language.

The SANE protocol can be run across any transport protocol that provides reliable data delivery. While SANE does not specify a specific transport protocol, it is expected that TCP/IP will be among the most commonly used protocols.

5.1 Data Type Encoding

5.1.1 Primitive Data Types

The four primitive types of the SANE standard are encoded as follows:

`SANE_Byte`**:** A byte is encoded as an 8 bit value. Since the transport protocol is assumed to be byte-orientd, the bit order is irrelevant.

`SANE_Word`**:** A word is encoded as 4 bytes (32 bits). The bytes are ordered from most-significant to least-significant byte (big-endian byte-order).

`SANE_Char`**:** A character is currently encoded as an 8-bit ISO LATIN-1 value. An extension to support wider character sets (16 or 32 bits) is planned for the future, but not supported at this point.

`SANE_String`**:** A string pointer is encoded as a `SANE_Char` array. The trailing NUL byte is considered part of the array and a `NULL` pointer is encoded as a zero-length array.

`SANE_Handle`**:** A handle is encoded like a word. The network backend needs to take care of converting these integer values to the opaque pointer values that are presented to the user of the network backend. Similarly, the SANE daemon needs to take care of converting the opaque pointer values it receives from its backends into 32-bit integers suitable for use for network encoding.

enumeration types**:** Enumeration types are encoded like words.

5.1.2 Type Constructors

Closely following the type constructors of the C language, the SANE network protocol supports the following four constructors:

pointer**:** A pointer is encoded by a word that indicates whether the pointer is a NULL-pointer which is then followed by the value that the pointer points to (in the case of a non-NULL pointer; in the case of a NULL pointer, no bytes are encoded for the pointer value).

***array*:** An array is encoded by a word that indicates the length of the array followed by the values of the elements in the array. The length may be zero in which case no bytes are encoded for the element values.

***structure*:** A structure is encoded by simply encoding the structure members in the order in which they appear in the corresponding C type declaration.

***union*:** A union must always be accompanied by a tag value that indicates which of the union members is the currently the active one. For this reason, the union itself is encoded simply by encoding the value of the currently active member.

Note that for type constructors, the pointer, element, or member values themselves may have a constructed type. Thus, the above rules should be applied recursively until a sequence of primitive types has been found.

Also SANE had no need for encoding of circular structures. This greatly simplifies the network protocol.

5.2 Remote Procedure Call Requests

The SANE network protocol is a client/server-style remote procedure call (RPC) protocol. This means that all activity is initiated by the client side (the network backend)—a server is restricted to answering requests sent by the client.

The data transferred from the client to the server is comprised of the RPC code (as a `SANE_WORD`), followed by arguments defined in the **request** column below. The format of the server's answer is given in the **reply** column.

5.2.1 `SANE_NET_INIT`

RPC Code: 0

This RPC establishes a connection to a particular SANE network daemon. It must be the first call in a SANE network session. The parameter and reply arguments for this call are shown in the table below:

request:	**reply:**
`SANE_Word version_code`	`SANE_Word status`
`SANE_String user_name`	`SANE_Word version_code`

The `version_code` argument in the request is the SANE version-code of the network backend that is contacting the network daemon (see Section 4.1). The "build-revision" in the version code is used to hold the network protocol version. The SANE network daemon receiving such a request must make sure that the network protocol version corresponds to a supported version since otherwise the encoding of the network stream may be incompatible (even though the SANE interface itself may be compatible). The `user_name` argument is the name of the user on whose behalf this call is being performed. If the network backend cannot determine a user-name, it passes a `NULL` pointer for this argument. No trust should be placed in the authenticity of this user-name. The intent of this string is to provide more convenience to the user. E.g., it could be used as the default-user name in subsequent authentication calls.

In the reply, `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values.[1] The `version_code` argument returns the SANE version-code that the network daemon supports. See the comments in the previous paragraph on the meaning of the build-revision in this version code.

5.2.2 SANE_NET_GET_DEVICES

RPC Code: 1

This RPC is used to obtain the list of devices accessible by the SANE daemon.

request:	**reply:**
`void`	`SANE_Word status`
	`SANE_Device ***device_list`

There are no arguments in the request for this call.

In the reply, `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values. The `device_list` argument is a pointer to a `NULL`-terminated array of `SANE_Device` pointers.

5.2.3 SANE_NET_OPEN

RPC Code: 2

This RPC is used to open a connection to a remote SANE device.

request:	**reply:**
`SANE_String device_name`	`SANE_Word status`
	`SANE_Word handle`
	`SANE_String resource`

The `device_name` argument specifies the name of the device to open.

In the reply, `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values. The `handle` argument specifies the device handle that uniquely identifies the connection. The `resource` argument is used to request authentication. If it has a non-`NULL` value, the network backend should authenticate the specified resource and then retry this operation (see Section 5.2.10 for details on how to authorize a resource).

[1]The sane network daemon should be careful not to leak information in the undefined portion of the reply.

5.2.4 SANE_NET_CLOSE

RPC Code: 3

This RPC is used to close a connection to a remote SANE device.

request:	**reply:**
`SANE_Word handle`	`SANE_Word dummy`

The `handle` argument identifies the connection that should be closed.

In the reply, the `dummy` argument is unused. Its purpose is to ensure proper synchronization (without it, a net client would not be able to determine when the RPC has completed).

5.2.5 SANE_NET_GET_OPTION_DESCRIPTORS

RPC Code: 4

This RPC is used to obtain *all* the option descriptors for a remote SANE device.

request:	**reply:**
`SANE_Word handle`	`Option_Descriptor_Array odesc`

The `handle` argument identifies the remote device whose option descriptors should be obtained.

In the reply, the `odesc` argument is used to return the array of option descriptors. The option descriptor array has the following structure:

```
struct Option_Descriptor_Array
  {
    SANE_Word num_options;
    SANE_Option_Descriptor **desc;
  };
```

5.2.6 SANE_NET_CONTROL_OPTION

RPC Code: 5

This RPC is used to control (inquire, set, or set to automatic) a specific option of a remote SANE device.

request:	**reply:**
`SANE_Word handle`	`SANE_Status status`
`SANE_Word option`	`SANE_Word info`
`SANE_Word action`	`SANE_Word value_type`
`SANE_Word value_type`	`SANE_Word value_size`
`SANE_Word value_size`	`void *value`
`void *value`	`SANE_String *resource`

The `handle` argument identifies the remote device whose option should be controlled. Argument `option` is the number (index) of the option that should be controlled. Argument `action` specifies what action should be taken (get, set, or set automatic). Argument `value_type` specifies the type of the option value (must be one of `SANE_TYPE_BOOL`, `SANE_TYPE_INT`, `SANE_TYPE_FIXED`, `SANE_TYPE_STRING`, `SANE_TYPE_BUTTON`). Argument `value_size` specifies the size of the option value in number of bytes (see Section 4.2.9 for the precise meaning of this value). Finally, argument `value` is a pointer to the option value. It must be a writeable area that is at least `value_size` bytes large. (Note that this area must be writable even if the action is to set the option value. This is because the backend may not be able to set the exact option value, in which case the option value is used to return the next best value that the backend has chosen.)

In the reply, argument `resource` is set to the name of the resource that must be authorized before this call can be retried. If this value is non-`NULL`, all other arguments have undefined values (see Section 5.2.10 for details on how to authorize a resource). Argument `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values. The `info` argument returns the information on how well the backend was able to satisfy the request. For details, see the description of the corresponding argument in Section 4.3.7. Arguments `value_type` and `value_size` have the same values as the arguments by the same name in corresponding request. The values are repeated here to ensure that both the request and the reply are self-contained (i.e., they can be encoded and decoded independently). Argument `value` is holds the value of the option that has become effective as a result of this RPC.

5.2.7 SANE_NET_GET_PARAMETERS

RPC Code: 6

This RPC is used to obtain the scan parameters of a remote SANE device.

request:	**reply:**
`SANE_Word handle`	`SANE_Status status`
	`SANE_Parameters params`

The `handle` argument identifies the connection to the remote device whose scan parameters should be returned.

In the reply, `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values. The argument `params` is used to return the scan parameters.

5.2.8 SANE_NET_START

RPC Code: 7

This RPC is used to start image acquisition (scanning).

request:	reply:
`SANE_Word handle`	`SANE_Status status`
	`SANE_Word port`
	`SANE_Word byte_order`
	`SANE_String resource`

The `handle` argument identifies the connection to the remote device from which the image should be acquired.

In the reply, argument `resource` is set to the name of the resource that must be authorized before this call can be retried. If this value is non-`NULL`, all other arguments have undefined values (see Section 5.2.10 for details on how to authorize a resource). Argument, `status` indicates the completion status. If the value is anything other than `SANE_STATUS_SUCCESS`, the remainder of the reply has undefined values. The argument `port` returns the port number from which the image data will be available. To read the image data, a network client must connect to the remote host at the indicated port number. Through this port, the image data is transmitted as a sequence of data records. Each record starts with the data length in bytes. The data length is transmitted as a sequence of four bytes. These bytes should be interpreted as an unsigned integer in big-endian format. The four length bytes are followed by the number of data bytes indicated by the length. Except for byte-order, the data is in the same format as defined for `sane_read()`. Since some records may contain no data at all, a length value of zero is perfectly valid. The special length value of `0xffffffff` is used to indicate the end of the data stream. That is, after receiving a record length of `0xffffffff`, the network client should close the data connection and stop reading data.

Argument `byte_order` specifies the byte-order of the image data. A value of 0x1234 indicates little-endian format, a value of 0x4321 indicates big-endian format. All other values are presently undefined and reserved for future enhancements of this protocol. The intent is that a network server sends data in its own byte-order and the client is responsible for adjusting the byte-order, if necessary. This approach causes no unnecessary overheads in the case where the server and client byte-order match and puts the extra burden on the client side when there is a byte-order mismatch. Putting the burden on the client-side improves the scalability properties of this protocol.

5.2.9 `SANE_NET_CANCEL`

RPC Code: 8

This RPC is used to cancel the current operation of a remote SANE device.

request:	reply:
`SANE_Word handle`	`SANE_Word dummy`

The `handle` argument identifies the connection whose operation should be cancelled.

In the reply, the `dummy` argument is unused. Its purpose is to ensure proper synchronization (without it, a net client would not be able to determine when the RPC has completed).

5.2.10 SANE_NET_AUTHORIZE

RPC Code: 9

This RPC is used to pass authorization data from the net client to the net server.

request:	reply:
SANE_String resource	SANE_Word dummy
SANE_String username	
SANE_String password	

The resource argument specifies the name of the resource to be authorized. This argument should be set to the string returned in the resource argument of the RPC reply that required this authorization call. The username and password are the name of the user that is accessing the resource and the password for the specified resource/user pair.

Since the password is not encrypted during network transmission, it is recommended to use the following extension:

If the server adds the string '$MD5$' to the resource-name followed by a random string not longer then 128 bytes, the client may answer with the MD5 digest of the concatenation of the password and the random string. To differentiate between the MD5 digest and a strange password the client prepends the MD5 digest with the string '$MD5$'.

In the reply, dummy is completely unused. Note that there is no direct failure indication. This is unnecessary since a net client will retry the RPC that resulted in the authorization request until that call succeeds (or until the request is cancelled). The RPC that resulted in the authorization request continues after the reply from the client and may fail with SANE_STATUS_ACCESS_DENIED.

5.2.11 SANE_NET_EXIT

RPC Code: 10

This RPC is used to disconnect a net client from a net server. There are no request or reply arguments in this call. As a result of this call, the connection between the client and the server that was established by the SANE_NET_INIT call will be closed.

Chapter 6

Contact Information

The SANE standard is discussed and evolved via a mailing list. Anybody with email access to the Internet can automatically join and leave the discussion group by sending mail to the following address.

`sane-devel-request@lists.alioth.debian.org`

To subscribe, send a mail with the body "`subscribe sane-devel`" to the above address.

A complete list of commands supported can be obtained by sending a mail with a subject of "`help`" to the above address. The mailing list is archived and available through the SANE home page at URL:

`http://www.sane-project.org/`

Index