IVI Driver ANSI C Specification

Version Number	Date of Version	Version Notes
1.0	February 9, 2026	First approved version

Abstract

This specification contains the ANSI C specific requirements for an IVI-ANSI-C driver, it is an IVI Language-Specific specification. This specification is to be used in conjunction with the IVI Driver Core specification.

Authorship

This specification is developed by member companies of the IVI Foundation. Feedback is encouraged. To view the list of member vendors or provide feedback, please visit the IVI Foundation website at www.ivifoundation.org.

Warranty

Trademarks

Table of Contents

Overview of the IVI-ANSI-C Driver Language Specification

This specification contains the ANSI C specific requirements for an IVI-ANSI-C driver; it is an IVI Language-Specific specification. This specification is to be used in conjunction with the IVI Driver Core specification.

This specification has several recommendations (identified by the use of the word *should* instead of *shall* in the requirement). These are included to provide a more consistent customer experience. However, in general, design decisions are left to the driver designer.

Relationship to the IVI Driver Core Specification

This specification contains ANSI C specific requirements for drivers that provide a library for use with ANSI C compilers or other clients that can use a compiled library to interface to an instrument.

This specification also requires that drivers comply with the IVI Driver Core specification. That specification is language independent and has several requirements that conformant drivers are required to satisfy independent of the driver language such as documentation, testing, and source code availability.

Relationship of IVI-ANSI-C to the IVI-C Specifications

This specification and other IVI Driver Core specifications have less extensive requirements to facilitate instrument interchangeability than IVI-C. For instance, IVI-ANSI-C drivers do not require the IVI Configuration Store.

However, there is no limitation to driver users utilizing both IVI-C and IVI-ANSI-C drivers in their system. Nor is there any inherent limitation to using an IVI-C driver in an ANSI C setting.

Drivers that conform to this specification do not automatically conform with IVI-C nor do IVI-C drivers automatically conform to this specification.

Substitutions

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the text that is used in place of the bracketed text. The *IVI Driver Core Specification* describes these substitutions.

The *IVI Driver Core Specification* uses the <DriverIdentifier> to indicate the name that uniquely identifies the driver. For IVI-ANSI-C drivers, the <DriverIdentifier> shall be constructed as:

- The first 2 characters shall designate the instrument manufacturer. These shall be one of the vendor abbreviations of the driver vendor assigned to the vendor in the IVI Specification VPP-9. Note that any vendor will be assigned an available 2-character abbreviation of their choice at no charge by the IVI Foundation. This abbreviation shall always be in upper case. If the instrument manufacturer is not registered in VPP-9, or if the driver supports multiple manufacturers' instruments the driver vendor may choose any two characters, however they shall include the final token indicating the driver vendor.
- The following characters shall identify the instrument models supported, and any other driver identifying information the driver vendor chooses. If a vendor expects multiple versions of a driver to be used in a single application, the vendor must differentiate the identifiers by incorporating the driver version into the vendor-provided string. These additional characters shall not include underscores ('_') due to the use of underscore to separate the <DriverIdentifier> from other identifiers.
- The final token indicates the driver vendor. If the driver vendor is the same as the instrument manufacturer, this token is not included.

Vendors should try to keep the <DriverIdentifier> short because it appears in any driver symbols that are put into a global namespace.

This document uses the following conventions regarding the <DriverIdentifier>:

- <DRIVER_IDENTIFIER> refers to the driver identifier in upper case separated from succeeding tokens with an underscore. The vendor abbreviations are NOT separated from the instrument model token with an underscore. An underscore is used to separate the driver identifier from the rest of the symbol. For instance, the token 'Foo' defined by vendor 'XY' for model family 'SigGen42' becomes: `XYSIGGEN42_FOO`.
- <driver_identifier> refers to the driver identifier in lower case, separated from succeeding tokens with underscores. The vendor abbreviation is NOT separated from the instrument model token name with an underscore. An underscore is used to separate the driver identifier from the rest of the symbol. For instance, the token 'Foo' defined by vendor 'XY' for model family 'SigGen42' becomes: `xysiggen42_foo`.

- <DriverIdentifier> is used when the context does not require further clarification, or when Pascal case is used. The vendor abbreviation is all upper case as is the first character of the model token. Unless stated otherwise, the driver identifier is separated from the rest of the symbol by putting the first character of the rest of the symbol in upper case. For instance, the token 'Foo' defined by vendor 'XY' for model family 'SigGen42' becomes: `XYSigGen42Foo`.

IVI-ANSI-C Driver Architecture

This section describes how IVI-ANSI-C instrument drivers use ANSI C. This section does not attempt to describe the technical features of ANSI C, except where necessary to explain a particular IVI-ANSI-C feature. This section assumes that the reader is familiar with ANSI C.

Operating Systems and Bitness

IVI-ANSI-C drivers can support various compilers and operating systems. Vendors that provide IVI-ANSI-C Drivers shall provide an IVI-ANSI-C driver for Microsoft Windows. The driver shall be supported on a version of Windows with a compiler that were both current when the driver was released or last updated. Driver vendors are encouraged to also support IVI-ANSI-C drivers on other operating systems and compilers important to their users.

In addition to the compliance documentation required by the IVI Driver Core specification (Compliance Documentation) IVI-ANSI-C drivers shall also document the compilers and compiler versions with which the driver has been tested and is supported.

Target ANSI C Versions

IVI-ANSI-C driver binaries (libraries) shall support clients using ISO/IEC 9899:1999 compilers (henceforth referred to as C99). Drivers are permitted to provide binaries with APIs that support newer ANSI C features, however they shall provide all instrument capabilities from C99 callable APIs.

> **Observation:**
>
> > Most modern C compilers support both C99 and C++. They typically distinguish between the two languages based on the file extension (`.c` vs. `.cpp` or `.cc`) or command-line options.

Header files for IVI-ANSI-C drivers shall be compilable by C99 compilers. If these compilers also support C++, then it is also recommended to allow the header files to be used in C++ applications. For example, use the `__cplusplus` preprocessor conditional to declare public API entry points with `extern "C"` linkage, so that clients using a C++ compiler will import the public API symbols using C name decorations (mangling), rather than C++ name decorations.

Public API entry points in IVI-ANSI-C driver binaries shall follow the compiler's C ABI (Application Binary Interface). This includes but is not limited to: name decorations, calling conventions, and structure packing/alignment.

When IVI-ANSI-C driver source code is provided it shall be compilable by C99 compilers, however drivers are encouraged to support newer versions of the ANSI C standards as well. This may require conditional compilation if other compilers are supported.

> **Observation:**

> > IVI-ANSI-C drivers that are not required to supply source code are not required to be implemented in C, although this spec requires that they be callable from C99.

IVI-ANSI-C Naming

To avoid naming collisions, symbols that the driver puts into a global name space shall be guaranteed unique by prefixing the symbol with an appropriately cased version of the <DriverIdentifier>.

In the following table, examples are all for a device with <DriverIdentifier> of `AC123Dev`, which would represent an instrument vendor identified by `AC` and a model or family identified by `123Dev`. The section Substitutions has detailed information on the driver identifier.

The following casing rules shall be followed:

Language Element	Example	Rule
function names	AC1234Dev_my_function	Function names shall be in snake case preceded by the <DriverIdentifier> and an underscore. Snake case is, words in lower case separated by underscores.
enumeration constants	AC123DEV_COLORS_TEAL	Enumeration constants shall be upper-case with underscore separators ('_') preceded by the <DRIVER_IDENTIFIER>. The enumeration constant name should be composed of the <DRIVER_IDENTIFIER> followed by a term identifying the enumeration type, and finally the constant name. That is: `<DRIVER_IDENTIFIER>_<ENUM_TYPE>_<CONSTANT>`.
`const` and macros (`#define`)	AC123DEV_MAX_POWER	`const` and macros shall be upper-case with underscore separators ('_') preceded by <DRIVER_IDENTIFIER>
types (`typedef`, `struct`, `enum`)	AC123DevMySpecialType	Types shall be in Pascal case (also known as upper camel-case) preceded by the <DriverIdentifier>. That is, words begin with upper case letter. Drivers should follow conventional exceptions for acronyms.
formal parameter names	start_frequency	Formal parameter names should be snake case. The driver identifier should not be used.

> **Observation:**
>
> > The IVI Foundation grants available 2-character vendor identifiers to any driver vendor requesting them at no cost. Assigned identifiers can be found in the current version of IVI VPP-9, referenced at the IVI Foundation Specification Download page.

> **Observation:**
>
> > Since each driver vendor is assigned a unique 2-character prefix, this scheme eliminates conflicts between driver vendors. Each vendor then manages the other characters in the

> > <DriverIdentifier> to eliminate collisions.

IVI-ANSI-C Filenames

Driver file names shall be snake case.

C source code files shall use a *.c* suffix, and C header files shall use a *.h* suffix.

Binary files should use filename suffixes conventional for the operating system and compilers they target.

In general IVI-ANSI-C drivers are composed of numerous files. The name of each file that is specific to the driver shall be prefixed by the <driver_identifier> in snake case.

If needed, vendors are permitted to include files with the driver that are common to multiple drivers provided by the vendor. These files do not require <driver_identifier> but shall be prefixed with the vendor 2-character VPP-9 prefix followed by other identifiers assigned by the driver vendor.

> **Observation:**
>
> > The provision to not require <driver_identifier> in the filename permits vendors to have common include files that define types that are common to several drivers. For instance, a struct for waveforms.

IVI-ANSI-C Data Types

Drivers should prefer the fundamental data types intrinsic to ANSI C and the types defined in ANSI C include files. This includes, but is not limited to: '<stdint.h>', '<float.h>' and '<string.h>'.

Drivers are encouraged to define other driver-specific types when ANSI types are not available. Vendors may find it beneficial to define types that are common to multiple drivers. However, drivers should avoid creating new types that are synonymous with those provided by ANSI.

Drivers shall define a driver-defined type for the driver session, thereby providing type-checking for that parameter. For details see the Session Parameter.

Drivers shall provide all include files necessary to use the driver in the driver package.

IVI-ANSI-C String Encoding

IVI-ANSI-C drivers' public APIs shall use UTF-8 string encoding.

> **Observation:**
>
> > The string encoding used to communicate with the instrument is instrument-specific. For performance reasons drivers are not required to validate the encoding of strings exchanged with the instrument.

IVI-ANSI-C Header Files

Drivers shall provide include files for their clients that contain:

- `#include` directives for any ANSI C include files required by the driver

- prototypes for all functions provided by the driver
- definitions for all types not provided by ANSI C that are needed by the client
- definitions of enumerated types and enumeration constants
- definitions of any macros used by the driver

Multiple Inclusion

All driver include files shall be protected against multiple inclusions. For instance, by defining a symbol when the file is first loaded, and subsequently bypassing included content on subsequent loads.

Example: A driver xysiggen42_sg_types.h, would use a compiler-specific pragma or define a symbol `XYSIGGEN42_SG_TYPES_H` when first loaded, then enclose the body of the .h file in appropriate `#ifdef` directives.

IVI-ANSI-C Function Style

The following subsections call out required IVI-ANSI-C style. There are additional rules and recommendations in repeated capabilities.

IVI-ANSI-C Function Naming

IVI-ANSI-C function names shall be snake case, but preceded with the <DriverIdentifier> in Pascal case followed by an underscore.

In general drivers are encouraged to organize ANSI C function names into a hierarchy. This is beneficial because:

- it provides helpful organization for customers to navigate complex APIs
- it permits vendors targeting drivers to both ANSI C and object-oriented languages
- it allows the documentation to be leveraged between ANSI C and object-oriented languages

Vendors should consider emulating the hierarchical API by appending the names of the elements of the hierarchy in generating the C identifier. For instance:

```
Dmm32.Measurement.Voltage.Span(start, stop);
```

Would translate into:

```
int32_t status = XYDmm32_measurement_voltage_span(Dmm32Session, start, stop);
```

> **Observation:**
>
> > As the user types the C identifier in a smart editor, the way that choices are presented to the user by the editor emulates a hierarchy.

The Session Parameter

As shown in the Base API drivers shall implement an *initialize* function that has an *out* parameter named `session`.

Drivers shall provide a `typedef` that specifies the type of `session`. The type defined by the `typedef` shall be named `<DriverIdentifier>Session`.

Drivers shall also provide a `const` that specifies a value that can be used as a sentinel to indicate an invalid session. The `const` shall be named `<DRIVER_IDENTIFIER>_INVALID_SESSION`.

All driver functions that reference a specific instance of the driver shall take `session` as the first parameter.

> **Observation:**
>
> > Driver designers frequently choose an integer for the session parameter which is used as an index to access the driver data. Driver designers also frequently choose to use a pointer type for the session parameter which directly points to the driver data. These and other approaches are permitted by these rules. Regardless, it is wise for the driver to take some steps to validate the session.

IVI-ANSI-C Status and Error Handling

All IVI-ANSI-C functions that may result in an error shall indicate errors to the client using the function return value. The return value shall be an *int32_t*. Zero shall be used to indicate success.

Negative return values shall indicate an error, positive return values shall indicate non-fatal warnings.

> **Observation:**
>
> > The driver function `<DriverIdentifier>_error_query()` is used to handle errors detected within the instrument that may not be indicated using the return value from functions.

Properties

Properties are values that clients can either get, set, or both.

Functions that get or set the value of the property shall have the word `get` or `set` as the last token of the function name.

The get/set functions shall:

- return a success code
- the first parameter shall be the driver session
- the next parameter(s) shall be repeated capability selector(s) if needed
- the final parameter shall be the value to set for set functions or a pointer to the value for get functions

Getting string properties is an exception and is handled using the variable sized data retrieval protocol.

> **Observation:**
>
> > Placing get/set at the *end* of the function name ensures that property accessors alphabetize appropriately into the function namespace.

Enumerated Types and Enumeration Constants

IVI-ANSI-C drivers may define enumerated types, which are integral types where the allowed values are specified by a set of named enumeration constants.

Enumerated type names shall be of the form `<DriverIdentifier><EnumeratedTypeName>` in Pascal case.

Enumeration constant names shall be of the form `<DRIVER_IDENTIFIER>_<ENUMERATED_TYPE_NAME>_<ENUMERATION_CONSTANT_NAME>` in upper case with underscores between words.

> **Observation:**
>
> > Enumeration constants are not scoped by the enumerated type to which they belong. Prefixing enumeration constants with the enumerated type name prevents name conflicts between constants.

The implementation of enumerated types is vendor-defined. Recommended implementations:

- A `typedef` for an `enum` type with corresponding enumeration constants.

  ```
  typedef enum {
      XYDMM32_FUNCTION_DC_VOLTS = 1,
      XYDMM32_FUNCTION_AC_VOLTS = 2
  } XYDmm32Function;
  ```

- A `typedef` for an integral type, with `#define`d enumeration constants.

  ```
  typedef uint32_t XYDmm32Function;
  #define XYDMM32_FUNCTION_DC_VOLTS (1)
  #define XYDMM32_FUNCTION_AC_VOLTS (2)
  ```

> **Observation:**
>
> > In C, the `enum` keyword must be specified when referring to a named enumerated type (e.g. `enum XYDmm32Function`). In C++, the `enum` keyword is optional and the enumerated type may be used directly (e.g. `XYDmm32Function`). Defining a `typedef` for an unnamed enumerated type allows referring to the type without specifying the `enum` keyword.

> **Observation:**
>
> > In C99, the underlying integral type of an `enum` is implementation-defined. Adding enumeration constants to an existing enumerated type may change the width and/or signedness of its underlying integral type, breaking binary and/or source compatibility. Popular C compilers for Windows and desktop Linux use 32-bit integers when possible, but embedded platforms or compiler options such as GCC's `-fshort-enums` may behave differently.

> > C23 and C++11 extend the `enum` syntax to allow specifying the underlying integral type. This is not legal C99 syntax, so drivers may only use this syntax if it is guarded with appropriate C preprocessor conditionals.
> >
> > Example:
> >
> > ```c
> > #if (defined(__STDC_VERSION__) && __STDC_VERSION__ >= 202311L) ||
> > (defined(__cplusplus) && __cplusplus >= 201103L)
> > #define XYDMM32_ENUM_TYPE : uint32_t
> > #else
> > #define XYDMM32_ENUM_TYPE
> > #endif
> >
> > typedef enum XYDMM32_ENUM_TYPE {
> >   XYDMM32_FUNCTION_DC_VOLTS = 1,
> >   XYDMM32_FUNCTION_AC_VOLTS = 2
> > } XYDmm32Function;
> > ```

> **Observation:**
>
> > In C99, enumerated types are not type-safe: an `enum` is compatible with its underlying integral type. As a result, enumerated types with the same underlying integral type may be used interchangeably. For example, if `XYDmm32Function` and `XYDmm32TempTransducerType` have the same underlying integral type, then you can pass `XYDMM32_FUNCTION_DC_VOLTS` to the `XYDmm32_temp_transducer_type_set` function without getting a compiler error or warning. In C++, enumerated types are type-safe, so compiling the same code with a C++ compiler would produce an error or warning.

If the sign of the enumerated type has no significance for the driver, drivers should prefer unsigned types.

> **Observation:**
>
> > These enumeration rules permit bit-mapped "flag" enumerations.

Variable Sized Data Retrieval Protocol

This section defines a *Variable Sized Data Retrieval Protocol* for the driver to return data to the client. This includes strings, arrays, and other dynamically sized structures. This protocol shall be used by functions that return variably sized data; however, it is not required for functions that have requirements that are not satisfied by the protocol.

IVI-ANSI-C requires that the driver client allocate memory for values provided by the driver in order to avoid difficulties related to the client freeing the memory after the buffer is no longer needed. Thus a consistent protocol is required for:

- the client to determine the required size of the buffer
- the client to determine the actual size of the returned value
- consistently passing the buffer and sizes

Driver authors are permitted to use other methods to negotiate buffer sizes with the client, if and only if they require functionality not provided by this approach. For instance:

- the function necessarily has side effects that preclude use of this protocol
- the read needs to return buffers of a client-specified length
- it is necessary for the driver to allocate the memory for the buffer

To implement this protocol, functions shall have the following parameters:

- `size` is an appropriately sized integer that indicates the size of the buffer allocated by the client. Under most circumstances, it should be of type `size_t`.
- `buffer` is a pointer to the client-allocated memory where the driver will write the data.
- `size_required` is a pointer to an integer of the same type and unit as `size` that shall be written by the driver. On a successful write to the buffer, the `size_required` shall indicate the quantity of data written to the buffer. If the buffer is not written, the driver uses this parameter to return the buffer size required for a subsequent call to the function.

The parameters should be presented in the function parameter list in the order listed above.

Driver writers are permitted to choose the formal parameter names. When choosing the formal parameter names, driver authors should consider the names above, however the term `buffer` should be replaced by a term that describes the value being returned.

For this protocol:

- `size` shall have units that correspond to the elements of the buffer. For instance, if the buffer is an array of 32-bit integers size shall be the number of 32-bit integers contained in the buffer, if the buffer is a string the units on size shall be the size of a *char*.
- If the function is called with either the *size* set to 0, or the *buffer* pointer set to *null* then the driver shall return the required size for the buffer in the `size_required` parameter and have no other side effects. In this case, the driver shall not return an error or warning.
- If the size parameter indicates the buffer is too small, the function shall indicate the size required in the `size_required` parameter and return an error. This call shall have no side effects.
- When the returned value is a string, the terminating null character shall be included in the length of the string, and it shall be written by the driver into the buffer.

This protocol should not be used if the function is returning data of known size such as a *struct*. In this case, the client should allocate the necessary memory (for instance by allocating a *struct*) and pass a pointer to it.

Repeated Capabilities

Repeated capabilities may be represented in two ways in IVI-ANSI-C drivers. Repeated capability instances may be specified by a method that selects the active instance (the *selector style*) or by selecting a particular instance using a function parameter or parameters. See the IVI Core Driver specification (Repeated Capabilities) for additional details.

Driver functions that require a repeated capability parameter(s) should pass it as the second (and or following) parameter(s), after the *session*.

If a driver implements multiple repeated capabilities (for instance N markers on M waveforms), the driver API may either define a scheme whereby multiple repeated capabilities can be passed in a single parameter, or it may use additional parameters.

For instance, if a driver chooses to use method parameters to identify N markers on M waveforms, it could:

- accept 2 parameters one for the marker and another for the waveform
- it could bitmap the parameters into an integer, so the second waveform's third marker may be identified as '0x23' (providing 4 bits for both parameters)
- it could accept a repeated capability structure defined by the driver
- it could treat the repeated capability identifiers as strings and pass a string such as "2:3"

Drivers are permitted to choose any of these, or use other approaches.

> **Observation:**
>
> > There are additional details on possible repeated capability implementation strategies in the IVI SCPI standard, the IVI Driver Core 'Using Strings as RepCap Selectors' Document, and the IVI Generation 2014 driver specifications.

> **Observation:**
>
> > Some applications requiring repeated capability parameters operate on multiple instances of a repeated capability at the same time. For these, bitmapping each repeated capability into an integer may work well.

> **Observation:**
>
> > A useful solution to the challenge of nested repeated capabilities is to treat them as a string with each element syntactically separated as in IVI-C and described in Using Strings as RepCap Selectors. The string is then parsed by the driver.

Documentation and Source Code

This specification does not have specific requirements on the format or distribution method of documentation and source code other than those called out in the IVI Driver Core Specification.

Thread Safety

IVI-ANSI-C drivers shall be thread-safe. That is, driver functions shall tolerate being called from foreign threads while the driver is currently executing on another thread.

Base IVI-ANSI-C API

This section gives a complete description of each function required for an IVI-ANSI-C Core driver. The following table shows the mapping between the required base driver APIs described in the IVI Driver Core specification (Required Driver APIs) and the corresponding IVI-ANSI-C specific APIs described in this section.

Required Driver API Mapping Table

Required Driver API (IVI Driver Core)	Core IVI-ANSI-C API

Required Driver API (IVI Driver Core)	Core IVI-ANSI-C API
Initialization	<DriverIdentifier>_init()
Driver Version	<DriverIdentifier>_driver_version_get()
Driver Vendor	<DriverIdentifier>_driver_vendor_get()
Error Query	<DriverIdentifier>_error_query()
Instrument Manufacturer	<DriverIdentifier>_instrument_manufacturer_get()
Instrument Model	<DriverIdentifier>_instrument_model_get()
Query Instrument Status Enabled (Get)	<DriverIdentifier>_query_instrument_status_enabled_get()
Query Instrument Status Enabled (Set)	<DriverIdentifier>_query_instrument_status_enabled_set()
Reset	<DriverIdentifier>_reset()
Simulate Enabled	<DriverIdentifier>_simulate_get
Supported Instrument Models	<DriverIdentifier>_supported_instrument_models_get()

IVI-ANSI-C Initialize Functions

The IVI-ANSI-C drivers shall implement two initialize functions, one which permits specifying options.

The parameters to the initialize functions are defined in the IVI Driver Core specification (Initialization). The following table shows their names and types for ANSI C:

Parameter	Description	Data Type
resource_name	Resource Name	const char *
id_query	ID Query	bool
reset	Reset	bool

The `<DriverIdentifier>_init_with_options()` function includes an *options* string used to specify initial settings and various configuration for the driver using name-value pairs. The format of the *options* string shall be: `<name1>=<value>;<name2>=<value>;...`. That is, the setting name is separated from the initial value with an equal sign ('=') and name value pairs are separated with semicolons (';').

For the initialization functions *simulation* is initially disabled unless specified otherwise by using the `<DriverIdentifier>_init_with_options()` function and specifying in the *options* string that *simulation* is enabled.

IVI-ANSI-C drivers may implement additional initialize functions.

Additional Required Functions for IVI-ANSI-C Drivers

This section defines additional required functions that are not specified in the IVI Driver Core specification.

Error Message Functions

IVI-ANSI-C drivers shall provide three functions to assist customers in interpreting errors and warnings encountered by the driver.

function	use
`<DriverIdentifier>_error_message`	this function converts an error returned by a driver function into a human-readable string
`<DriverIdentifier>_last_error_message`	this alternative to `<DriverIdentifier>_error_message` returns the message for the most recent error
`<DriverIdentifier>_clear_last_error_message`	this function clears the last error message

The following paragraphs specify the operation of these functions:

- `<DriverIdentifier>_error_message` returns a fixed string that describes the return value from a driver function. The return value may indicate an error, a warning or no error. For *no error* the driver shall return an empty string. To use this function, the client passes the return value from a driver function and a string buffer using the standard IVI-ANSI-C buffer protocol. A human readable string that describes the error or warning is returned. If the passed error code is not defined by the driver, the driver shall return an appropriate error code and not modify the string buffer.

- `<DriverIdentifier>_last_error_message` returns a string indicating the most recent error from the driver. This function may provide more detailed error information than `<DriverIdentifier>_error_message`. Subsequent errors overwrite the buffer used by this function. Calling this function does not clear its internal buffer. To clear the last error, call `<DriverIdentifier>_clear_last_error_message`.

- `<DriverIdentifier>_clear_last_error_message` clears the buffer used by `<DriverIdentifier>_last_error_message`. If there are no intervening errors, a subsequent call to `<DriverIdentifier>_last_error_message` shall return an empty string indicating no error.

Read and Clear Error Queue

IVI-ANSI-C Drivers for instruments that implement an error queue shall provide the `<DriverIdentifier>_read_and_clear_error_queue` function. `<DriverIdentifier>_read_and_clear_error_queue` reads as much of the instrument error queue as possible and formats it into the provided buffer. If the instrument error queue length exceeds what can we written into the buffer, the function shall put as many complete formatted errors into the buffer as possible and return success.

The `<DriverIdentifier>_read_and_clear_error_queue` function provides an alternative to using the `<DriverIdentifier>_error_query` function specified in the IVI Driver Core specification (Error Query).

SCPI instruments include both an integer and a string in the error queue, therefore for each entry taken from the queue the integer is formatted into the string, followed by a comma (','), and then the error message from the instrument enclosed in double quotes (""). Each error is separated by semicolons. Only complete error entries are written into the string. The string itself shall be null terminated.

`<DriverIdentifier>_read_and_clear_error_queue` does not follow the standard Variable Sized Data Retrieval Protocol because the function is unable to determine the size required for the buffer without performing a destructive read of the error queue. Therefore, clients must allocate a buffer large enough to capture a sufficient number of errors for their application. The allocated buffer must include space for the trailing null.

`<DriverIdentifier>_read_and_clear_error_queue` shall return an error if the *size* parameter is zero or the *error_string* pointer is null. Note that this differs from the standard Variable Sized Data Retrieval Protocol.

For instance, the following would be a valid string produced by this function for a *SCPI* instrument: '-131,"Invalid Suffix";-200,"Execution Error";-210,"Trigger Error";-220,"Parameter Error"'.

> **Observation:**
>
> > To implement this, the function should read successive entries from the error queue, formatting them into the buffer. Once an entry is retrieved that does not entirely fit into the buffer, that entry and any successive entries should be read from the instrument and discarded.

Prototypes of Required Driver Functions

In the prototypes below:

1. The *<DriverIdentifier>* is inserted per the rules in substitutions.
2. *<DriverIdentifier>Session* is the type specified in the session parameter.

```
/* Initialization functions */
int32_t <DriverIdentifier>_init(const char *resource_name, bool id_query,  bool
reset, <DriverIdentifier>Session* session_out);
int32_t <DriverIdentifier>_init_with_options(const char* resource_name, bool
id_query, bool reset, const char* options, <DriverIdentifier>Session*
session_out);

/* Functions specified in the base spec */
int32_t <DriverIdentifier>_driver_version_get(<DriverIdentifier>Session session,
size_t size, char* version_out, size_t* size_required);
int32_t <DriverIdentifier>_driver_vendor_get(<DriverIdentifier>Session session,
size_t size, char* vendor_out,  size_t* size_required);
int32_t <DriverIdentifier>_error_query(<DriverIdentifier>Session session, int32_t*
error_code_out, size_t size, char* error_message_out, size_t* size_required);
int32_t <DriverIdentifier>_instrument_manufacturer_get(<DriverIdentifier>Session
session, size_t size, char* manufacturer_out, size_t* size_required);
int32_t <DriverIdentifier>_instrument_model_get(<DriverIdentifier>Session session,
char* model_out);
int32_t
<DriverIdentifier>_query_instrument_status_enabled_get(<DriverIdentifier>Session
session, bool* instrument_status_enabled_out);
int32_t
<DriverIdentifier>_query_instrument_status_enabled_set(<DriverIdentifier>Session
session, bool instrument_status_enabled);
int32_t <DriverIdentifier>_reset(<DriverIdentifier>Session session);
int32_t <DriverIdentifier>_simulate_get(<DriverIdentifier>Session session, bool*
simulate_out)
```

```
int32_t
<DriverIdentifier>_supported_instrument_models_get(<DriverIdentifier>Session
session, size_t size, char* supported_instrument_models_out, size_t*
size_required)

/* Additional functions required for driver error management */
int32_t <DriverIdentifier>_error_message(int32_t error, size_t size, char
*error_message, size_t* size_required);
int32_t <DriverIdentifier>_last_error_message(<DriverIdentifier>Session session,
size_t size, char *error_message, size_t* size_required);
int32_t <DriverIdentifier>_clear_last_error(<DriverIdentifier>Session session);

/* Additional function for working with the instrument error queue */
int32_t <DriverIdentifier>_read_and_clear_error_queue(<DriverIdentifier>Session
session, size_t size, char *error_queue);
```

ANSI C Specific Notes (see IVI Driver Core specification (Required Driver APIs) for general requirements):

- Drivers are permitted to implement a set function on `Simulate`. However, if they do so, they shall properly manage the driver state when turning simulation on and off.

Direct IO functions

Per the *IVI Driver Core specification*, IVI Drivers for instruments that have an ASCII command set such as SCPI shall also provide an API for sending messages to and from the instrument over the ASCII command channel. This section specifies those functions.

In the following '<hierarchy>' indicates whatever hierarchy path the driver designer chooses for the direct IO functions.

Required Driver API (IVI Driver Core)	**Core IVI-ANSI-C API**
I/O Timeout Set	`<DriverIdentifier>_<hierarchy>_timeout_milliseconds_set()`
I/O Timeout Get	`<DriverIdentifier>_<hierarchy>_timeout_milliseconds_get()`
Read Bytes	`<DriverIdentifier>_<hierarchy>_read_bytes()`
Read String	`<DriverIdentifier>_<hierarchy>_read_string()`
Write Bytes	`<DriverIdentifier>_<hierarchy>_write_bytes()`
Write String	`<DriverIdentifier>_<hierarchy>_write_string()`

Prototypes for Direct IO Functions

```
int32_t <DriverIdentifier>_<hierarchy>_timeout_milliseconds_set(const void*
session, const long);
int32_t <DriverIdentifier>_<hierarchy>_timeout_milliseconds_get(const void*
session, &long);
int32_t <DriverIdentifier>_<hierarchy>_iosession_get(const void* session, void
**iosession);    // Optional
int32_t <DriverIdentifier>_<hierarchy>_read_bytes(const void* session, const long size, uint8_t *);
int32_t <DriverIdentifier>_<hierarchy>_read_string(const void* session,const long size, char *);
int32_t <DriverIdentifier>_<hierarchy>_write_bytes(const void* session, const long size, const uint8_t *);
int32_t <DriverIdentifier>_<hierarchy>_write_string(const void* session, const char *);
```

Notes:

- The *optional* `iosession` read-only property should return a session for the underlying IO library.
- The Direct IO read functions are unable to use the Variable Sized Data Retrieval Protocol because they have no a priori knowledge of the transfer size.

> **Observation:**
>
> > Drivers should consider including a query function that combines read and write.

Packaging Requirements for ANSI C

IVI-ANSI-C drivers are provided as packages. The packaging technology should be appropriate for the driver user for instance: zip, tar, or self-extracting archives.

Driver packages shall include:

- static and dynamic libraries for the supported OS/compilers
- include files (.h) for the supported OS/compilers
- driver license terms
- the IVI Compliance document per the IVI Driver Core Specification
- README.md file
- documentation or a README file that indicates how to acquire documentation
- documentation or a README file that indicates how to acquire source code for drivers that are required to provide source code

The IVI Driver Core specification (README.md) has detailed requirements on the *README.md* file and the *IVI Compliance document* as well as examples. All of these requirements shall be followed by IVI-ANSI-C drivers.

Driver packages may include additional files at the discretion of the provider. They may also organize the files into subdirectories at their discretion.

> **Observation:**
>
> > Driver providers may need to provide separate packages for different compilers on a given platform. For instance, a provider may have separate Windows driver packages for *Microsoft Visual C/C++* and *gcc*.

Signing

Where possible, the driver library files should be signed by the vendor.

IVI-ANSI-C Driver Conformance

IVI-ANSI-C Drivers are required to conform to all the rules in this document as well as the rules in the IVI Driver Core specification. They are also required to be registered on the IVI website.

Drivers that satisfy these requirements are IVI-ANSI-C drivers and may be referred to as such.

Registered conformant drivers are permitted to use the IVI Conformant Logo.

Driver Registration

Driver providers wishing to obtain and use the IVI Conformance logo shall submit to the IVI Foundation the driver compliance document described in *IVI Driver Core Specification*, Section Driver Conformance along with driver information and a point of contact for the driver. The information shall be submitted to the IVI Foundation website. Complete upload instructions are available on the site. Driver vendors who submit compliance documents may use the IVI Conformant logo graphics.

The IVI Foundation may make some driver information available to the public for the purpose of promoting IVI drivers. All information is maintained in accordance with the IVI Privacy Policy, which is available on the IVI Foundation website.