

SNiFF+TM

Version 3.2 for Unix and Windows

Parser Development Kit



TakeFive Software, Inc.
Cupertino, CA
E-mail: info@takefive.com

TakeFive Software GmbH
5020 Salzburg, Austria
E-mail: info@takefive.co.at

Copyright

Copyright © 1992–1999 TakeFive Software Inc.

All rights reserved. TakeFive products contain trade secrets and confidential and proprietary information of TakeFive Software Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure.

Parts of SNIFF+:

Copyright 1991, 1992, 1993, 1994 by Stichting Mathematisch Centrum,
Amsterdam, The Netherlands.

Portions copyright 1991-1997 Compuware Corporation.

Trademarks

SNIFF+ is a trademark of TakeFive Software Inc.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Credits

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

Table of Contents

SNiFF+ Parser Development Kit	5
Introduction	5
Basics	6
Creating and editing a .Isd file for non C/C++ language constructs	8
Editing the new .Isd file	8
Command Reference	10
Data types	10
Job description	10
Data Structures	12
Job control functions	15
Connection to SNiFF+	15
Get a compile job	16
Terminate a compile job	17
Close connection to SNiFF+	18
Semantic actions	18
Comment	20
Macro definition	21
Include statement	22
Start class declaration	23
End class declaration	24
Base class specifier	26
Enumerator	28
Enumerator item	29
Variable or member variable declaration	30
Function declaration/definition	31
Type definition	34
Function parameter	36
User-defined symbol	38
Template argument	39
Symbol reference	41
Utilities	42
Reset	42
Error message	43

SNiFF+ Parser Development Kit

Introduction

The SNiFF+ Parser Development Kit allows you to write parsers for SNiFF+ which either replace or work in parallel with the default C/C++ parser. The API comes in the form of a linkable library. The functions in this library manage low-level communications with SNiFF+ and provide an interface for sending information about semantic constructs extracted by the parser. Although the terminology used by the functions is based on C/C++, the functions can be used for writing a parser for any language which implements concepts compatible to those in C/C++. The SNiFF+ Parser Development Kit contains functions with C linkage.

Who should be reading this document

This document should be read by anyone who intends to use the SNiFF+ Parser Development Kit. It is assumed that readers of this document are familiar with SNiFF+ and its functionality.

SNiFF+ version

You will need SNiFF+3.0 or newer to use the SNiFF+ Parser Development Kit with SNiFF+.

For SNiFF+ 3.2:

You need the latest SNiFF+ Parser Development Kit which is delivered with SNiFF+ 3.2, i.e., you cannot use the Parser Development Kit of SNiFF+ 3.0 or 3.1 with SNiFF+ 3.2. Recompile your old parsers with the new Parser Development Kit in order to use your old parser.

Your Feedback

Your feedback is always welcome. If there are aspects of this document that are unclear to you, or if you have any questions or comments concerning the SNiFF+ Parser Development Kit, please contact us at one of the following e-mail addresses:

Europe:

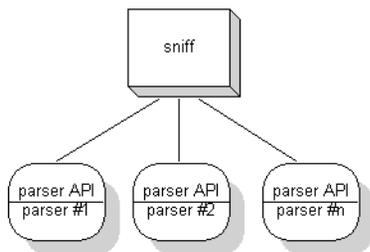
sniff-support@takefive.co.at

USA:

sniff-support@takefive.com

Basics

The ability to work with different parsers simultaneously makes SNIFF+ a truly multi-platform, multi-language development environment. Either a single parser can be used for projects which contain only source files of a specific language, or several parsers can be simultaneously employed for handling multi-language projects seamlessly. All project files can be edited, browsed and managed within a consistent user interface, resulting in higher productivity and efficiency. Multiple parsers can run simultaneously, they directly connect to the `sniff` core process.



Parsing is handled by SNIFF+ in the following way:

- When parsing a file, SNIFF+ loads the appropriate parser as specified by the settings in the **File Types** view of the *Project Attributes* dialog window.
- SNIFF+ sends a request to the parser for each source file which needs to be parsed.
- The parser provides SNIFF+ with symbol information for the source file.
- The parser then waits for the next parsing request. It is deactivated by SNIFF+ on termination of its associated `sniff` process.

Symbol types

The following symbol types and other language elements are currently available:

comment	macro definition
include statement	class declaration
member variable definition	member function declaration and definition
global function definition	global variable definition
type definition	enumerator definition
user-defined symbols	

The parser also delivers information about global symbol references for function definitions. This information is used by the Cross Referencer. Currently, symbol types must be mapped to existing C/C++ symbol types.

2

Creating and editing a .lsd file for non C/C++ language constructs

Creating the .lsd file

SNiFF+ reads the .lsd files in your `$SNIFF_DIR/config/parser` directory to find out how it should name the language constructs that it comes across when parsing your source files. You can edit these files to reflect the names and abbreviations of the language constructs that your parsers will parse.

To do so, please complete the following steps (in the instructions and comments that follow, the term “your language” is used to mean the language that your parser parses):

- Make a copy of `$SNIFF_DIR/config/parser/template.lsd` and rename it to `<name_of_your_parser>.lsd`
- (`<name_of_your_parser>.lsd` should also be in `$SNIFF_DIR/config/parser`.)
- Load `<name_of_your_parser>.lsd` into an editor.

Editing the new .lsd file

Editing parameters

Caution!

The .lsd file contains parameters that are used by SNiFF+ to map the language constructs in your language to those in C/C++. Before you begin modifying this part, there are a couple of things that you should keep in mind:

- Do **not** delete any of the parameters or add new parameters.
- Do **not** change the order of the parameters.
- There are a number of parameters that begin with `"res`. **Do not modify these parameters.**
- Do **not** modify the parameter `"User Defined Symbol" : "UserDef"`. This parameter is used by SNiFF+ for your user-defined symbols. SNiFF+ automatically knows what type of language construct a user-defined symbol is when your parser comes across the user-defined symbol and calls the semantic action function `User-defined symbol`.

- There is no limit to the length of names and abbreviations that you give to language constructs in your language. However, we suggest that you keep them short. Note that all names and abbreviations must be alphanumeric strings (without any tabs or spaces separating characters of a string).

Parameters

In the list of parameters, you can see the parameter that determines the name of the entry in the **Language** drop-down menu (e.g., in the Symbol Browser) for your language:

```
"Language Name" : "Ansi C/C++"
```

- By default, the name of the entry in the **Language** drop-down menu is `Ansi C/C++`. Change the name of this entry to the name of the your language.
- The remaining parameters in the parameter list refer to the names and abbreviations of the language constructs in your language. Let's look at one of these parameters:

```
"Enum" : "enum"
```

- The string in quotes in the first column refers to the C/C++ language construct (here: `Enum`). **Do not modify this string.**
- The string in quotes in the second column specifies how the C/C++ language construct is mapped to a language construct in your language. Replace this string with the name of a language construct in your language.
- You may not need to use all of the parameters in the parameter list. For those parameters that you do not need, replace the second string in quotes with `--empty--`.

For example, when SNIFF+ comes across the following parameter, it will ignore it:

```
"Const" : "--empty--"
```

There are a number of parameters that begin with `Short of`. These parameters specify the abbreviated names of language constructs. For example, the abbreviated name of the C/C++ language construct `Enum` is `en`, as specified in the following parameter:

```
"Short of Enum" : "en"
```

Abbreviations of language constructs are used in the Cross Referencer tool.

- Select an abbreviation for those language constructs in your language that either reference or are referenced by other language constructs.
- Save the file. The parameters in the file will become effective the next time you launch SNIFF+.

3

Command Reference

This API Reference is subdivided into the following sections:

- [Data types — page 10](#)
- [Job control functions — page 15](#)
- [Semantic actions — page 18](#)
- [Utilities — page 42](#)

Data types

Job description

`pi_job`

SYNOPSIS

```
struct
{
    char        source_file_name;
    pi_bool     cpp_on;
    char        *cpp_define;
    char        *cpp_include;
    char        *parser_config_file;
    char        *generate_dir;
    pi_bool     no_generate;
    pi_bool     generate_only;
};
```

DESCRIPTION

Describes a compile job.

<code>source_file_name</code>	full pathname of the source file to be parsed
<code>cpp_on</code>	preprocessor switch (see project attributes)
<code>cpp_define</code>	preprocessor defines (see project attributes)
<code>cpp_include</code>	preprocessor includes (see project attributes)
<code>parser_config_file</code>	full pathname of the parser configuration file
<code>generate_dir</code>	SNiFF+ internal usage
<code>no_generate</code>	SNiFF+ internal usage
<code>generate_only</code>	SNiFF+ internal usage

The SNiFF+ Parser Development Kit fills in this data structure as per the current parsing request.

See also [pi_getjob — page 16](#)

Data Structures

pi_data

SYNOPSIS

```
struct
{
    int    from;
    int    to;
    int    from2;
    int    to2;
    int    argstart;
    int    argend;
    int    ctorstart;
    int    ctorend;
    char   *name;
    char   *type;
    char   *classname;
    char   *nameSpace;
    int    flags;
    pi_id  related;
};
```

DESCRIPTION

Describes a data structure. This data structure is used by all semantic action functions as a parameter which describe the details of a symbol item. It has to be allocated and filled in by the parser application.

<code>from</code>	offset relative to file begin
<code>to</code>	offset relative to file begin
<code>from2</code>	offset relative to file begin
<code>to2</code>	offset relative to file begin
<code>argstart</code>	not supported in this version
<code>argend</code>	not supported in this version
<code>ctorstart</code>	not supported in this version
<code>ctorend</code>	not supported in this version
<code>name</code>	see semantic actions for details
<code>type</code>	see semantic actions for details
<code>classname</code>	see semantic actions for details
<code>nameSpace</code>	scope of symbol, see semantic actions for details
<code>flags</code>	see semantic actions for details
<code>related</code>	NOT USED in this version

Please refer to the specific semantic action functions for a description of the data members used by these functions.

See also all semantic action functions and the `pi_reset()` function

Flags used in `pi_data.flags`

<code>PI_VIRTUAL</code>	virtual function
<code>PI_ABSTRACT</code>	pure virtual function
<code>PI_INLINE</code>	inline function
<code>PI_CONST</code>	const variable or function

PI_PRIVATE	private member access or inheritance
PI_PUBLIC	public member access or inheritance
PI_PROTECTED	protected member access or inheritance
PI_TEMPLATE	template class
PI_FRIEND	friend class or function
PI_STATIC	static variable or function
PI_DEFAULT_VAL	function parameter has default value
PI_FUNCTION	function or member function
PI_VARIABLE	variable or member variable
PI_MACRO	macro definition
PI_TYPEDEF	type definition
PI_ENUM	enumeration type
PI_ENUM_ITEM	enumeration item
PI_CLASS	class declaration
PI_STRUCT	structure declaration
PI_UNION	union declaration
PI_DEFINITION	definition of a function or member function
PI_DECLARATION	declaration of a member function
PI_READACCESS	read access to a variable or member variable
PI_WRITEACCESS	write access to a variable or member variable
PI_STDINCL	if is standard <include>, else "include"
PI_FINAL	if symbol is final in JAVA
PI_SYNCHRONIZED	if symbol is synchronized in JAVA
PI_NATIVE	if symbol is native in JAVA
PI_TRANSIENT	if symbol is transient in JAVA
PI_VOLATILE	if symbol is volatile

Return values of semantic action functions

PI_NOVALUE	-1	default return value (void)
PI_BADID	-2	illegal id in <code>pi_data.related</code>
PI_NOJOB	-3	semantic action called without active job

Job control functions

Connection to SNIFF+

`pi_init`

SYNOPSIS

```
pi_bool pi_init(pi_bool remoteParser, char *parserId);
```

PARAMETER

`remoteParser`: FALSE - The parser and SNIFF+ are running on the same machine
 TRUE - The parser is running on a different machine than SNIFF+.

`char *parserId`: A string that identifies the parser to SNIFF+. It must be the base-name of the parser executable without extension (e.g., for the executable `/usr/local/sniff/bin/myparser` or `c:\coolstuff\myparser.exe`, the `parserid` is the string `myparser`).

DESCRIPTION

Establishes a connection to SNIFF+. This function must be called once at parser start-up time before all other API function calls.

RETURN VALUE

FALSE	Connection to SNIFF+ successfully established
TRUE	Cannot connect to SNIFF+

Get a compile job

`pi_getjob`

SYNOPSIS

```
pi_job *pi_getjob();
```

PARAMETER

None

DESCRIPTION

Get a new compile job. When SNIFF+ requests the parsing of a file, this function returns the compile job. It waits for the next parsing request from SNIFF+ before returning a further value.

RETURN VALUE

> NULL	All details of a compile job. (see also description of <code>struct pi_job</code> for details)
= NULL	No more compile jobs. SNIFF+ requests the parser to exit.

Terminate a compile job

```
pi_endjob()
```

SYNOPSIS

```
pi_bool pi_endjob(pi_job *job);
```

PARAMETER

`pi_job *job` Pointer to `pi_job` returned by previous `pi_getjob()` call.

DESCRIPTION

Terminate a compile job. This function must be called after parsing a file; e.g., after calling the last semantic action function.

RETURN VALUE

<code>FALSE</code>	Job successfully terminated.
<code>TRUE</code>	Error during termination. Caused by a SNIFF+/parser internal synchronization error.

Note

`pi_getjob()` and `pi_endjob()` functions must be called alternately.

Close connection to SNIFF+

```
pi_terminate
```

SYNOPSIS

```
pi_bool pi_terminate();
```

PARAMETER

None

DESCRIPTION

Close connection to SNIFF+. This function must be called before exiting the parser.

RETURN VALUE

FALSE	Connection successfully closed
TRUE	Error when closing connection. Caused by a SNIFF+/parser internal synchronization error.

Semantic actions

Note

All functions return `PI_NOVALUE` in this version. The data member, `pi_data.related`, is not used in this version.

Introduction

All fields in this section have the following structure

- Semantic action
- Synopsis
- Parameter
- C/C++ example
- SNIFF+ usage

each of the above is explained in more detail below.

SYNOPSIS

```

type of semantic action      semantic action function
function:

```

PARAMETER

Relevant members of the `pi_data` structure pertaining to this function call. In the pages that follow, the parameter list, `{x | y | z}`, is a “one of”, meaning one of the three options (separated by the pipe symbol) must be declared. The parameter list, `(x|y|z)`, prompts you to declare one of the options when the semantic action function is called within a class scope, otherwise none.

DESCRIPTION

Indicates when the function should be called. The description also includes:

C/C++ example

Given for each specific language construct (except Comment). Whenever this construct is parsed, the semantic action function is called. Your parser should call this function if it recognizes this or a similar language construct.

Each example is followed by a description of how SNIFF's Fuzzy Parser would parse it. For example:

```

pi_data.from:                points to first character of the name “foo”
pi_data.to:                  points to last character of the name “foo”
pi_data.name:                “foo”
pi_data.flags:               PI_CLASS

```

SNiFF+ usage

If your parser recognizes this or a similar language construct, you should notice this in your SNiFF+ application. Language constructs highlighted in the Source Editor are done so between the coordinates `pi_data.from` and `pi_data.to` (found under **PARAMETER**), which are character positions relative to the beginning of the file currently being parsed.

Comment

SYNOPSIS

```
pi_id      pi_comment(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start comment
<code>pi_data.to:</code>	end comment
<code>pi_data.related:</code>	NOT USED (future use: bind comment to any object)

DESCRIPTION

This function must be called whenever the parser detects a comment in the source file.

SNiFF+ usage:
highlighted in Source Editor

Macro definition

SYNOPSIS

```
pi_id pi_macro(pi_data *);
```

PARAMETER

```
pi_data.from: start macro name
pi_data.to: end macro name
pi_data.name: macro name
pi_data.type: macro value
```

DESCRIPTION

This function must be called whenever the parser detects a macro definition in the source file.

C/C++ example

```
#define FLAG 0x01
```

SNiFF+ Fuzzy Parser action

Each example is followed by a description of how SNiFF+'s Fuzzy Parser would parse it. For example:

```
pi_data.from: points to first character of the name "FLAG"
pi_data.to: points to last character of the name "FLAG"
pi_data.name: "FLAG"
pi_data.type: "0x01"
```

SNiFF+ usage

macro name highlighted in the Source Editor and shown in the Symbol Browser (as a macro)

Include statement

SYNOPSIS

```
pi_id pi_include(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start of " " or < > include statement
<code>pi_data.to:</code>	end of " " or < > include statement
<code>pi_data.name:</code>	name of the file not including " " or < > delimiters
<code>pi_data.flags:</code>	PI_STDINCL

DESCRIPTION

This function must be called whenever the parser detects an include statement in the source file. Parameters `pi_data.from` and `pi_data.to` are needed for the SNIFF+ Include Browser to function properly.

C/C++ example

```
#include <foo.h>
```

SNIFF+ Fuzzy Parser action

<code>pi_data.from:</code>	points to "<"
<code>pi_data.to:</code>	points to ">"
<code>pi_data.name:</code>	"foo.h"
<code>pi_data.flags:</code>	PI_STDINCL

SNIFF+ usage

highlighted in the Source Editor, includes shown in Include Browser, dependency generation for make support

Start class declaration

SYNOPSIS

```
pi_id      pi_start_class(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start class name
<code>pi_data.to:</code>	end class name
<code>pi_data.name:</code>	class name
<code>pi_data.flags:</code>	{PI_CLASS PI_STRUCT PI_UNION}, PI_TEMPLATE, PI_FRIEND
<code>pi_data.related:</code>	<code>pi_id</code> of class nesting this class Default: previous class non-terminated class declaration
<code>pi_data.classname:</code>	full string of classname including scope e.g., X.Y.Z. Class (The delimiter is a full stop)
<code>pi_data.nameSpace:</code>	full scope of class e.g., X.Y.Z

DESCRIPTION

This function must be called whenever the parser detects a start class declaration in the source file. When `pi_start_class` is called, a new class declaration scope is opened. All subsequent calls to `pi_function`, `pi_variable`, `pi_enum` and `pi_typdef` indicate member declarations of this class. The function `pi_end_class()` closes the class declaration scope. Class declaration scopes can be nested.

C/C++ example

```
class foo { /*...*/ };
```

SNiFF+ Fuzzy Parser action

```

pi_data.from:      points to first character of the name "foo"
pi_data.to:        points to last character of the name "foo"
pi_data.name:      "foo"
pi_data.flags:     PI_CLASS

```

SNiFF+ usage

highlighted in Source Editor, listed in Symbol and Class Browsers, class hierarchy built in Hierarchy Browser

Note

The base classes of this class can be specified by calling `pi_base()` immediately after calling `pi_start_class`.

See also `pi_end_class`, `pi_base`, `pi_function`, `pi_variable`, `pi_enum`, `pi_typedef`

End class declaration

SYNOPSIS

```
pi_id pi_end_class(pi_data *);
```

PARAMETER

```

pi_data.from:      start position of the whole class declaration
pi_data.to:        end position of the whole class declaration
pi_data.related:   pi_id of class to be terminated
                   Default: previous class non-terminated class declaration

```

DESCRIPTION

This function must be called whenever the parser detects an end class declaration in the source file.

C/C++ example

```
class foo { /*...*/ };
```

SNiFF+ Fuzzy Parser action

`pi_data.from:` points to the opening brace (after the classname)

`pi_data.to:` points to the closing brace

`pi_data.name:` "foo"

SNiFF+ usage

highlighted in Source Editor

See also `pi_start_class`

Base class specifier

SYNOPSIS

```
pi_id pi_base(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start base class name
<code>pi_data.to:</code>	end base class name
<code>pi_data.name:</code>	base class name, optionally including preceding scope specifiers separated by “?”
<code>pi_data.flags:</code>	(<code>PI_PUBLIC</code> <code>PI_PROTECTED</code> <code>PI_PRIVATE</code>), <code>PI_VIRTUAL</code>
<code>pi_data.related:</code>	<code>pi_id</code> of class having this base class Default: previous class non-terminated class declaration
<code>pi_data.classname:</code>	full string of classname including scope e.g., X.Y.Z. Class (The delimiter is a full stop)
<code>pi_data.nameSpace:</code>	full scope of class e.g., X.Y.Z

DESCRIPTION

This function specifies one base class of the current class declaration. It must be called whenever the parser detects a base class specifier in the source file.

C/C++ example

```
class X : public Base_Class_Of_X { ... };
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:      points to start of "Base_Class_Of_X"  
pi_data.to:       points to end of "Base_Class_Of_X"  
pi_data.name:     "Base_Class_Of_X"  
pi_data.flags:    PI_PUBLIC
```

SNiFF+ usage

class hierarchy built in Hierarchy Browser, inheritance information shown in Class Browser

Note

The function `pi_base()` must be called immediately after calling `pi_start_class()`.
See also `pi_start_class`

Enumerator

SYNOPSIS

```
pi_id pi_enum(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start enumerator name
<code>pi_data.to:</code>	end enumerator name
<code>pi_data.name:</code>	enumerator name
<code>pi_data.flags:</code>	(<code>PI_PUBLIC</code> <code>PI_PROTECTED</code> <code>PI_PRIVATE</code>), <code>PI_STATIC</code>
<code>pi_data.related:</code>	<code>pi_id</code> of class including the enumerator as member Default: previous class non-terminated or global scope if the function is called outside of a class declaration

DESCRIPTION

This function must be called whenever the parser detects an enumerator in the source file.

C/C++ example

```
enum Boolean { false, true };
```

SNiFF+ Fuzzy Parser action

<code>pi_data.from:</code>	points to start of "Boolean"
<code>pi_data.to:</code>	points to end of "Boolean"
<code>pi_data.name:</code>	"Boolean"

SNiFF+ usage

highlighted in Source Editor

Enumerator item

SYNOPSIS

```
pi_id      pi_enum_item(pi_data *);
```

PARAMETER

```
pi_data.from:      start enumerator item name
pi_data.to:        end enumerator item name
pi_data.name:      enumerator item name
pi_data.related:   pi_id of enumerator containing this item
                  Default: previous enumerator
```

DESCRIPTION

This function must be called for each enumeration item of the enumerator.

C/C++ example

```
enum Boolean { false, true }; // "false" and "true" are
                             // enumeration items of Boolean
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:      points to first character of the enumeration item (e.g.,
                  "false")
pi_data.to:        points to last character of the enumeration item (e.g.,
                  "false")
pi_data.name:      e.g., "false"
```

Note

`pi_enum_item` must be called twice: both for "false" and "true"

SNiFF+ usage

highlighted in Source Editor, listed in the Symbol Browser

Note

The function `pi_enum_item()` must be called immediately after calling `pi_enum()`.

Variable or member variable declaration

SYNOPSIS

```
pi_id pi_variable(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start variable name
<code>pi_data.to:</code>	end variable name
<code>pi_data.name:</code>	variable name
<code>pi_data.flags:</code>	(<code>PI_PUBLIC</code> <code>PI_PROTECTED</code> <code>PI_PRIVATE</code>), <code>PI_STATIC</code> , <code>PI_CONST</code>
<code>pi_data.type:</code>	variable type
<code>pi_data.related:</code>	<code>pi_id</code> of class including the variable as member Default: previous class not terminated or global scope if no non-terminated class found

DESCRIPTION

This function must be called whenever the parser detects a variable or member variable declaration in the source file. The declared variable is a member variable if the function `pi_variable()` is called inside a class declaration (between `pi_start_class()` and `pi_end_class()`).

C/C++ example

```
static char ch; // ch is a variable of data type char
```

SNiFF+ Fuzzy Parser action

```

pi_data.from:    points to first character of "ch"
pi_data.to:     points to last character of "ch"
pi_data.name:   "ch"
pi_data.flags:  PI_STATIC
pi_data.type:   char

```

SNiFF+ usage

highlighted in Source Editor, listed in the Symbol Browser

Function declaration/definition

SYNOPSIS

```
pi_id pi_function(pi_data *);
```

PARAMETER

pi_data.from:	start function name (including class name if member function)
pi_data.to:	end function name
pi_data.from2:	start function implementation if inline
pi_data.to2:	end function implementation in inline
pi_data.name:	function name (exclusive class name)
pi_data.flags:	{PI_DECLARATION PI_DEFINITION},
(PI_PUBLIC PI_PROTECTED PI_PRIVATE), PI_STATIC, PI_INLINE, PI_VIRTUAL, PI_CONST, PI_TEMPLATE, PI_FRIEND, PI_ABSTRACT	
pi_data.type:	return type

<code>pi_data.classname:</code>	class name if member function (required only if <code>PI_DEFINITION</code> is set)
<code>pi_data.nameSpace:</code>	full scope of class e.g., X.Y.Z
<code>pi_data.related:</code>	<code>pi_id</code> of class including the function as member Default: previous class not terminated or global scope if no non-terminated class found

DESCRIPTION

This function must be called whenever the parser detects a function declaration/definition in the source file. The declared function is a member function if the function `pi_function()` is called inside a class declaration (between `pi_start_class()` and `pi_end_class()`) and `PI_DECLARATION` is set. If `PI_DEFINITION` is set, `pi_data.classname` specifies the container class. If `pi_data.classname` is not set and `PI_DEFINITION` is set, a global function definition is assumed.

C/C++ example

Example 1

```
int bar( int ) { ... }
```

SNiFF+ Fuzzy Parser action

<code>pi_data.from:</code>	points to start of "bar"
<code>pi_data.to:</code>	points to end of "bar"
<code>pi_data.from2:</code>	points to the opening brace (following "bar")
<code>pi_data.to2:</code>	points to the closing brace
<code>pi_data.name:</code>	"bar"
<code>pi_data.flags:</code>	<code>PI_DEFINITION</code>
<code>pi_data.type:</code>	<code>int</code>
<code>pi_data.classname:</code>	<code>NULL; //default value</code>

SNiFF+ usage

highlighted in Source Editor, listed in the Symbol Browser

Example 2

```
class bar
{ ...
    get()
}
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:      points to start of "get"
pi_data.to:        points to end of "get"
pi_data.name:      "get"
pi_data.flags:     PI_DECLARATION
pi_data.classname: NULL; //default value
```

SNiFF+ usage

highlighted in Source Editor, listed in the Symbol and Class Browsers

Note

The `pi_function` call must be made between `pi_start_class` and `pi_end_class` of "bar"

Example 3:

```
bar::get()
{
    ...
}
```

SNiFF+ Fuzzy Parser action

```

pi_data.from:           points to start of "get"
pi_data.to:             points to end of "get"
pi_data.name:           "get"
pi_data.flags:          PI_DEFINITION
pi_data.classname:     "bar"

```

SNiFF+ usage

highlighted in Source Editor, listed in the Symbol and Class Browsers

Type definition

SYNOPSIS

```

pi_id      pi_typedef(pi_data *);

```

PARAMETER

```

pi_data.from:      start typedef name
pi_data.to:        end typedef name
pi_data.name:      typedef name
pi_data.flags:     (PI_PUBLIC | PI_PROTECTED | PI_PRIVATE)
pi_data.type:      type definition
pi_data.related:   pi_id of class including the typedef as member
                   Default: previous class not terminated or global scope if no non-
                   terminated class found

```

DESCRIPTION

This function must be called whenever the parser detects a type definition in the source file.

C/C++ example

```

typedef char *string;

```

SNiFF+ Fuzzy Parser action

```
pi_data.from:      points to start of "string"  
pi_data.to:       points to end of "string"  
pi_data.name:     "string"  
pi_data.type:     char*
```

SNiFF+ usage

highlighted in Source Editor

Note

The defined type is valid in a class scope if the function `pi_typedef()` is called inside a class declaration (between `pi_start_class()` and `pi_end_class()`).

Function parameter

SYNOPSIS

```
pi_id pi_parameter(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start parameter name
<code>pi_data.to:</code>	end parameter name
<code>pi_data.name:</code>	parameter name
<code>pi_data.flags:</code>	PI_DEFAULT_VAL
<code>pi_data.type:</code>	parameter type
<code>pi_data.related:</code>	<code>pi_id</code> of function definition/declaration or reference Default: previous function definition/declaration or reference

DESCRIPTION

This function must be called whenever the parser detects a function parameter in the source file. The call of `pi_function()` must immediately be followed by calls of `pi_parameter()`. The `pi_comment()`, `pi_error_msg()` and `pi_reset()` functions are the only ones which may be called between `pi_function()` and `pi_parameter()` calls. `pi_function()` can also be used to specify the current parameter list of a function reference (see `pi_reference()` below).

C/C++ example

```
int min(int v1, char v2) // v1 and v2 are the function
                        // parameters (arguments)
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:    points to start of parameter name (e.g., v1)
pi_data.to:      points to end of parameter name (e.g., v1)
pi_data.name:    e.g., "v1"
pi_data.type:    e.g., int
```

Note

`pi_parameter` must be called twice: both for `v1` and `v2`

SNiFF+ usage

highlighted in Source Editor

Note

`pi_data.related` is not used in this version!

User-defined symbol

SYNOPSIS

```
pi_id pi_userdef(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start symbol name
<code>pi_data.to:</code>	end symbol name
<code>pi_data.name:</code>	symbol name
<code>pi_data.type:</code>	type is the string that the Symbol Browser uses for the symbol type in the Type drop-down. By using different values for the <code>pi_data.type</code> parameter, you can create as many user-defined symbol types in SNIFF+ as you want. Note that you can also have multiple symbols with the same <code>pi_data.type</code> parameter. In this case, you will see all of these symbols in the Symbol List of the Symbol Browser when you choose type from the Type drop-down.
<code>pi_data.related:</code>	not supported! Default: previous function definition/declaration or reference

DESCRIPTION

This symbol allows you to create new entries in the SNIFF+ Symbol Table.

SNIFF+ usage

highlighted in Source Editor, listed in Symbol Browser

Note

`pi_data.related` is not used in this version!

Template argument

SYNOPSIS

```
pi_id pi_template_argument(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start argument name
<code>pi_data.to:</code>	end argument name
<code>pi_data.name:</code>	argument name
<code>pi_data.related:</code>	<code>pi_id</code> of class definition or function definition/declaration or parameter having this template argument Default: previous class definition or function definition/declaration or parameter

DESCRIPTION

This function must be called whenever the parser detects a template argument in the source file.

C/C++ example

```
template <class Type> // class Type is the argument
                    // (parameter) of the template
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:      points to start of "class Type"
pi_data.to:        points to end of "class Type"
pi_data.name:      "class Type"
```

SNiFF+ usage

highlighted in Source Editor

Note

This function is not implemented in any of the API versions.

Symbol reference

SYNOPSIS

```
pi_id pi_reference(pi_data *);
```

PARAMETER

<code>pi_data.from:</code>	start position of the symbol reference relative to the <code>pi_data.from</code> position of the related (previous) <code>pi_function()</code> call
<code>pi_data.name:</code>	name of the symbol referred
<code>pi_data.classname:</code>	class name if the symbol is class member
<code>pi_data.nameSpace:</code>	only if symbol is a class member, full scope of class e.g., X.Y.Z
<code>pi_data.flags:</code>	{PI_FUNCTION PI_VARIABLE PI_MACRO PI_TYPEDEF PI_ENUM PI_ENUM_ITEM PI_STRUCT PI_CLASS}, PI_READACCESS, PI_WRITEACCESS
<code>pi_data.related:</code>	<code>pi_id</code> of function definition or declaration referring to the symbol Default: previous function definition/declaration

DESCRIPTION

This function must be called whenever the parser detects a function which refers to a symbol, e.g. function call, variable access, macro usage, et cetera. The references are used by the Cross Referencer tool. The flags `PI_READACCESS` and `PI_WRITEACCESS` must be set if the referred symbol is a variable (`PI_VARIABLE`). The function `pi_reference()` must be called immediately after the function definition, which consists of a call to `pi_function()` where the `PI_DEFINITION` flag is set, optionally followed by `pi_parameter()` calls. In case of a function reference (`PI_FUNCTION` is set), the actual parameter list can be specified. This can be done by calling `pi_parameter()` immediately after calling `pi_reference()`.

C/C++ example

```
foo()
{
bar();
}
```

SNiFF+ Fuzzy Parser action

```
pi_data.from:    points to start of "bar"
pi_data.name:    "bar"
pi_data.flags    PI_FUNCTION
```

SNiFF+ usage

shown in Xref tool

Note

If the referenced function is a member of a class, it must also be declared in the project (call of `pi_function()` (PI_DECLARATION)). Otherwise, SNiFF+ cannot show references made to this member function.

Utilities

Reset

SYNOPSIS

```
void    pi_reset(pi_data *);
```

DESCRIPTION

Initializes the `pi_data` structure. This function must be called before setting data members of the `pi_data` structure. The function does **NOT** free memory pointed by the members on `pi_data`, but the pointers will be set to NULL.

Caution

Only change the value of those data members that you want to send to SNiFF+. Do not change the value of the other data members!

Error message

SYNOPSIS

```
pi_bool pi_error_msg(char *);
```

DESCRIPTION

Send an error message to SNIFF+. The messages appear in the Log window after calling `pi_endjob()`. The sequence `@B` in the message toggles boldface printing.

Note:

You can indicate the type of error message being displayed in the Log window by using one of the following four letters as the first letter of the error message:

Letter	Description
W	Warning
F	Fatal error
E	Error
I	Information

If you use one of these four letters, it's corresponding description appears in the Log window before the name of the message sender. If you do not use one of these four letters, the string "Message from" appears instead.

