STAPIN API Documentation

1.1. Overview

API documentation for STAPIN Symbol Table API for Pin Library. STAPIN uses TCF and its API, here is the link to TCF API:

1.2. Namespaces

1.2.1. stapin

STAPIN namespace

1.3. Enumerations

1.3.1. E_context_update_method

Methods for updating the CPU context.

FULL	The entire CPU context including floating point state and registers is set
FAST	Only general purpose registers, IP, SP, and flags are set
IP_ONLY	Only the Instruction Pointer (IP) is set

1.3.2. E_symbol_type

Types of symbols.

UNKNOWN	The symbol type is unknown.
CLASS	The symbol represents a class.
FUNCTION	The symbol represents a function.
INTEGER	The symbol represents an integer.
POINTER	The symbol represents a pointer.
REAL	The symbol represents a real number (floating point).
ARRAY	The symbol represents an array.
VALUE	The symbol represents a value.

1.3.3. E_symbol_flags

Flags for symbols.

LOCATION_FETCH_ERROR	There was an error fetching the location of the symbol.
VALUE_IN_MEM	The value of the symbol is stored in memory.
VALUE_IN_REG	The value of the symbol is stored in a register.
OPTIMIZED_AWAY	The symbol has been optimized away.
SYMBOL_IS_TYPE	The symbol represents a type.
DW_STACK_VALUE	The symbol is a stack value.
IS_BIT_FIELD	The symbol is a bit field.

1.3.4. E_expression_type

Types of expressions.

UNKNOWN	The expression type is unknown.
UNSIGNED	The expression represents an unsigned value.
SIGNED	The expression represents a signed value.
DOUBLE	The expression represents a double precision floating point value.
CLASS	The expression represents a class.
FUNCTION	The expression represents a function.
POINTER	The expression represents a pointer.
ARRAY	The expression represents an array.
ENUM	The expression represents an enumeration.

1.4. Structures

1.4.1. Source_loc

Source location information.

const char*	srcFileName	The name of the source file.
ADDRINT	startAddress	The starting address of the source location.
int32_t	startLine	The starting line number of the source location.
int32_t	startColumn	The starting column number of the source location.
ADDRINT	endAddress	The ending address of the source location.
int32_t	endLine	The ending line number of the source location.
int32_t	endColumn	The ending column number of the source location.
ADDRINT	nextAddress	The address of the next source location.
ADDRINT	nextStmtAddress	The address of the next statement.

1.4.2. Symbol

Symbol information.

const char*	name	The name of the symbol.
const char*	uniqueId	A unique identifier for the symbol.
const char*	typeUniqueId	A unique identifier for the symbol's type. If not applicable, it is an empty string (not null).
const char*	baseTypeUniqueId	A unique identifier for the symbol's base type. If not applicable, it is an empty string (not null).
E_symbol_type	type	The type of the symbol, relating to its data type.
size_t	size	The size of the symbol.
ADDRINT	memory	The memory address of the symbol if it resides in memory. If the symbol is in a register, this is the index for the register.
REG	reg	The register where the symbol is stored if it resides in a register. If not, it is null.

E_symbol_flags	flags	Flags that relate to the memory placement of the symbol.

1.4.3. Expression

Expression information.

const char*	name	The name of the expression (the expression itself).
const char*	typeUniqueID	A unique identifier for the expression's type. If not applicable, it is an empty string (not null).
const char*	parentName	The name of the parent expression. If there is no parent, it is an empty string (not null).
const char*	symbolUniqueID	A unique identifier for the symbol representing the expression. If not applicable, it is an empty string (not null).
E_expression_type	typeFlag	The type flag of the expression.
int	level	The level of the field: 0 indicates the main parent, higher values indicate deeper nested fields.
uint16_t	size	The size of the expression.
uint8_t	data [EXPRESSION_DATA_BYTES]	The data of the expression, up to EXPRESSION_DATA_BYTES bytes.

1.5. Functions

1.5.1. Initialization and Notification

```
bool init();
```

Use this to initialize STAPIN.

Returns
true if STAPIN was successfully initialized, false otherwise. If STAPIN initialization failed any call to STAPIN API will fail.

```
bool notify_image_load(IMG img);
```

Notify STAPIN of an image load event. This function must be called from an Image Load Callback for every image the caller wishes to inspect using STAPIN.

Parameters
img: The image of interest

Returns
true If the image was correctly loaded by STAPIN, false otherwise. If this function fails for a particular image then trying to use STAPIN to inspect code from that image will fail.

```
bool notify_image_unload(IMG img);
```

Notify STAPIN of an image unload event. This function must be called from an Image Unload Callback for every image registered with **notify_image_load**.

Parameters
img: The image of interest

Returns
true If the image was successfully unregistered.
false If the image was not previously registered with STAPIN.

```
bool notify_thread_start(const CONTEXT* ctx, THREADID threadId);
```

Notify STAPIN of a new application thread. This function must be called from a Thread Start Callback for every thread the caller wishes to use STAPIN to inspect running code in the given thread context.

Parameters
ctx: Pin's context at thread start. The context will be set for the current thread as by a call to **set_context** with **E_context_update_method::FULL**.
threadId: Pin's thread Id for the new thread

Returns
true If thread registration succeeded, false otherwise. If thread registration failed for a given thread then further calls to STAPIN for that thread will fail.

```
bool notify_thread_fini(THREADID threadId);
```

Notify STAPIN of application thread exit. This function must be called from a Thread Fini Callback for every thread registered with **notify_thread_start**.

Parameters
threadId: The thread Id for the exiting thread.

Returns
true If the thread was successfully unregistered.
false If the thread was not previously registered with STAPIN.

```
bool notify_fork_in_child(const CONTEXT* ctx, THREADID threadId);
```

Notify STAPIN that fork occurred.

Parameters
ctx: Pin's context after fork in child. The context will be set for the current thread as by a call to **set_context** with **E_context_update_method::FULL**.
threadId: Pin's thread Id for child after fork

Returns
true If child registration succeeded, false otherwise. If child process registration failed for a given child then further calls to STAPIN for that child will fail.

1.5.2. Context Management

```
bool set_context(const CONTEXT* ctx, E_context_update_method contextUpdateMethod = E_context_update_method::FULL);
```

Set the context for future STAPIN queries. The given context will be used for the current thread until the next call to **set_context**.

Parameters
ctx: The Pin context under which STAPIN should operate for the current thread.
contextUpdateMethod: The context update method. The default is **E_context_update_method::FULL**.

Returns
true If the context for the current thread was set successfully, false otherwise. If the function failed, then the last context successfully set will be used.

1.5.3. Source Location

```
size_t get_source_locations(ADDRINT startAddress, ADDRINT endAddress, Source_loc* locations, size_t locCount);
```

Get an array of source locations for the given address range.

Parameters
startAddress: The start of the address range to get source location information for.
endAddress: The end of the address range to get source location information for.
locations: A pointer to an array that can receive the source location information for the given address range.
locCount: The number of entries in the array. This is the maximum number of locations that will be returned in the array even if there are more locations for the given address range.

Returns
The number of source locations filled inside **locations**. If this value is 0, it means that no source code locations were found for the given address.

Note
With optimized code, it is possible that multiple source locations map to this same address even if **startAddress** is equal to **endAddress**.

1.5.4. Symbol Management

Note
Whenever fetching symbols (get_* or find_*), users must call **set_context** before with the relevant context.

```
Symbol_iterator* get_symbols(const Symbol* parent = nullptr);
```

Get an iterator that can be used to go over symbols visible in the current context.

Parameters
parent: This argument can be used to specify a symbol scope. For instance if parent is a symbol of a function then the iterator will allow iterating over the function arguments and return type (first child is return type, arguments follow in left to right order). If the symbol belongs to a structure or a class, the iterator will allow iterating members. If nullptr is passed then an iterator to the symbols visible in the current context will be returned. The default is nullptr.

Returns
A symbol iterator or nullptr if an error occurred.

```
void reset_symbols_iterator(Symbol_iterator* iterator);
```

Reset the iterator to the beginning.

Parameters
iterator: The symbol iterator to reset.

```
Symbol_iterator* find_symbol_by_name(const char* symbolName, const Symbol* parent = nullptr);
```

Get an iterator that can be used to go over symbols with the same name visible from the current context.

Parameters
symbolName: The symbol name to search for in the current context and parent scope.
parent: This argument can be used to specify a symbol scope. For instance if parent is a symbol of a function then the iterator will allow iterating over the function arguments and return type (first child is return type, arguments follow in left to right order). If the symbol belongs to a structure or a class, the iterator will allow iterating members. If nullptr is passed then an iterator to the symbols visible in the current context will be returned. The default is nullptr.

Returns
A symbol iterator or nullptr if an error occurred.

```
bool get_next_symbol(Symbol_iterator* iterator, Symbol* symbol);
```

Get the next symbol object from a symbol iterator.

Parameters
iterator: The symbol iterator from which to bring the next symbol.
symbol: The next symbol information.

Returns
true If the next symbol was retrieved.
false If there are no more symbols in the iterator or an error occurred.

```
void close_symbol_iterator(Symbol_iterator* iterator);
```

Close and release a symbol iterator opened by **get_symbols** or **find_symbol_by_name**.

Parameters
iterator: The symbol iterator to close and release resources for.

Note
Users must call this function whenever opening a symbol iterator (using **get_symbols** or **find_symbol_by_name**.

```
bool get_symbol_by_address(ADDRINT address, Symbol* symbol);
```

Fill the symbol information for the symbol belonging to a given address.

Parameters
address: The address of the symbol in the current context.
symbol: The symbol belonging to this address.

Returns
true If symbol information was retrieved, false otherwise.

Note
Not all symbols may have an address associated with them. For instance, a symbol may reside in a register for a given context. See **get_symbol_by_reg**.

```
bool get_symbol_by_reg(REG reg, Symbol* symbol);
```

Fill the symbol information for the symbol belonging to a given register.

Parameters
reg: The register of the symbol in the current context.
symbol: The symbol belonging to this register.

Returns
true If symbol information was retrieved, false otherwise.

bool get_symbol_by_id(const char* uniqueId, Symbol* symbol);

Fill the symbol information for the symbol belonging to a given unique Id.

Parameters
uniqueId: The unique Id of the symbol.
symbol: The symbol belonging to this unique Id.

Returns
true If symbol information was retrieved, false otherwise.

```
bool get_rtn_symbol(RTN rtn, Symbol* symbol);
```

Fill the symbol information for the symbol belonging to a given routine specified by a Pin RTN.

Parameters
rtn: The routine (RTN) the symbol in the current context.
symbol: The symbol belonging to this routine.

Returns
true If symbol information was retrieved, false otherwise.

1.5.5. Expression Management

Note
Whenever fetching expressions (using **evaluate_expression**), users must call **set_context** before with the relevant context.

```
Expression_iterator* evaluate_expression(const std::string& expression);
```

Get an iterator that can be used to go over expressions that got evaluated.

Parameters
expression: This argument is the expression to be evaluated.

Returns
An expression iterator or nullptr if an error occurred.

```
void reset_expressions_iterator(Expression_iterator* iterator);
```

Reset the iterator to the beginning.

Parameters
iterator: The expression iterator to reset.

```
bool get_next_expression(Expression_iterator* iterator, Expression* expression);
```

Get the next expression object from an expression iterator.

Parameters
iterator: The expression iterator from which to bring the next expression.
expression: The next expression information.

Returns
true If the next expression was retrieved.
false If there are no more expressions in the iterator or an error occurred.

```
void close_expression_iterator(Expression_iterator* iterator);
```

Close and release an expression iterator opened by **evaluate_expression**.

Parameters
iterator: The expression iterator to close and release resources for.

Note
Users must call this function whenever opening an expression iterator (using **evaluate_expression**).