Compiler Specification

Table of Contents

Overview

The *compiler specification* is a required part of a Ghidra language module for supporting disassembly and analysis of a particular processor. Its purpose is to encode information about a target binary which is specific to the compiler that generated that binary. Within Ghidra, the SLEIGH specification allows the decoding of machine instructions for a particular processor, like Intel x86, but more than one compiler can produce those instructions. For a particular target binary, understanding details about the specific compiler used to build it is important to the reverse engineering process. The compiler specification fills this need, allowing concepts like parameter passing conventions and stack mechanisms to be formally described.

A compiler specification is a single file contained in a module's `data/languages` directory with a ".cspec" suffix. There may be more than one ".cspec" file in the directory, if Ghidra supports multiple compilers for the same processor. The compiler specification is identified by the 5th field of Ghidra's *processor id*. The id is explicitly linked with the ".cspec" by adding a tag in the root ".ldefs" file for the processor, also in the same directory.

Example 1.

Defining the processor id `x86:LE:32:default:gcc` and associating it with the file `x86gcc.cspec`

```
<language_definitions>
  ...
  <language processor="x86"
            endian="little"
            size="32"
            variant="default"
            version="2.3"
            slafile="x86.sla"
            processorspec="x86.pspec"
            manualindexfile="../manuals/x86.idx"
            id="x86:LE:32:default">
    <description>Intel/AMD 32-bit x86</description>
    <compiler name="Visual Studio" spec="x86win.cspec" id="windows"/>
    <compiler name="gcc" spec="x86gcc.cspec" id="gcc"/>
    <compiler name="Borland" spec="x86borland.cspec" id="borland"/>
  </language>
  ...
</language_definitions>
```

A compiler specification is just an XML file, so it needs to start with the usual XML directive and it always has `<compiler_spec>` as the root XML tag. All specific compiler features are described using subtags to this tag. In principle, all the subtags are optional except the `<default_prototype>` tag, but there is generally a minimum set of tags that are needed to create a useful specification (See ???). In general, the subtags can appear in any order. The only exceptions are that tags which define names, like `<prototype>`, must appear before other tags which use that name.

The rest of this document is broken up into sections that roughly correspond with aspects of compiler design, and then subsections within these address particular tags.

Varnode Tags

Many parts of the compiler specification use tags that describe a single varnode. Since architectures frequently name many of their registers or special memory locations, it is convenient for the specification designer to be able to use these names. But in some cases there is no name and the designer must fall back on the defining triple for a varnode: an *address space*, an *offset* and a *size*. Hence there are really two different XML tags that are used to describe varnodes and both are referred to as a **varnode tag**.

The `<register>` tag is used to specify formally named registers, usually defined by the SLEIGH specification for the processor. The name must be given in a *name* attribute for the tag.

The `<varnode>` tag is used to generically describe any varnode. It must take three attributes: *space* is a formal name of the address space containing the varnode, *offset* is an unsigned integer specifying the byte offset of the varnode within the space, and *size* is an integer specifying the size of the varnode in bytes. The `<varnode>` tag can be used to describe any varnode, including named registers, global RAM locations, and stack locations. For stack locations, the offset is interpreted relative to the function that is being decompiled or is otherwise in scope. An offset of 0, for instance typically refers to the memory location on the stack being pointed to by the formal stack pointer register, upon entry to the function being analyzed.

Example 2.

```
  <register name="EAX"/>
  <register name="r1"/>
```

```
<varnode space="ram" offset="0x1020" size="4"/>
<varnode space="stack" offset="8" size="8"/>
<varnode space="stack" offset="0xfffffff8" size="2"/>
<varnode space="register" offset="0" size="1"/>
```

Compiler Specific P-code Interpretation

<context_data>

Table 1.

Attributes and Children	
`<context_set>`	(0 or more) Set a context variable across a region of memory
`<tracked_set>`	(0 or more) Set default value of register

A `<context_data>` tag consists of zero or more `<context_set>` and `<tracked_set>` subtags, which allow certain values to be assumed by analysis.

<context_set>

Table 2.

Attributes and Children		
`space`		Name of address space
`first`		(Optional) Starting offset of range
`last`		(Optional) Ending offset of range
`<set>`		Specify the context variable and the new value
	`name`	Name of the context variable
	`val`	Integer value being set
	`description`	(Optional) Description of what is set

A `<context_set>` tag sets a SLEIGH context variable over a specified address range. This potentially affects how instructions are disassembled within that range. This is more commonly used in the *processor specification* file but can also be used here for specific compilers. The attributes `space`, `first`, and `last` describe the range. Omitting `first` and/or `last` causes the range to start at the beginning and/or run to the end of the address space respectively. The `<set>` subtag describes the variable and its setting.

Example 3.

```
<context_data>
  <context_set space="ram">
    <set name="mode16" val="1" description="Set 16-bit mode across all of ram"/>
  </context_set>
</contextdata>
```

<tracked_set>

Table 3.

Attributes and Children		
`space`		Name of address space
`first`		(Optional) Starting offset of range
`last`		(Optional) Ending offset of range
`<set>`		Specify the register and the new value
	`name`	Name of the register
	`val`	Integer value being set
	`description`	(Optional) Description of what is set

A `<tracked_set>` tag informs the decompiler that a register takes a specific value for any function whose entry point is in the indicated range. Compilers sometimes know or assume that registers have specific values coming into a function it produces. This tag allows the decompiler to make the same assumption and possibly use constant propagation to make further simplifications.

Example 4.

```
<context_data>
  <tracked_set space="ram">
    <set name="spsr" val="0"/>
  </tracked_set>
</context_data>
```

<callfixup>

Table 4.

Attributes and Children		
`name`		The identifier for this callfixup
`<target>`		(0 or more) Map this callfixup to a specific symbol
	`name`	The specific symbol name
`<pcode>`		Description of p-code to inject.

<pcode>

Table 5.

Attributes and Children		
`paramshift`		(Optional) Integer for shifting parameters at the callpoint.
`<body>`		P-code to inject.
	`text`	

Compilers frequently make use of special bookkeeping functions that are really internal to the compiler and not a direct reflection of functions in the original source code. During analysis it can be helpful to replace a call to such a function with a snippet of p-code that inlines the behavior, or a portion of the behavior, so that the decompiler can use it during its simplification rather than displaying it as an opaque call. A typical use is to inline *prologue* functions that help set up a stack frame.

The `name` attribute can be used to identify the callfixup within the Ghidra CodeBrowser and manually force certain functions to be replaced. The `name` attribute of the `<callfixup>` tag and any optional `<target>` subtags identify function names which will *automatically* be replaced.

The text of the `<body>` subtag is fed directly to the SLEIGH semantic expression parser to create the p-code snippet. Identifiers are interpreted as formal registers, if the register exists, but are otherwise interpreted as temporary registers in the *unique* space of the processor. Its usually best to surround text with the XML <![CDATA[construct.

Example 5.

```
<callfixup name="get_pc_thunk_bx">
  <target name="__i686.get_pc_thunk.bx"/>
  <pcode>
    <body><![CDATA[
    EBX = * ESP;
    ESP = ESP + 4;
    ]]></body>
  </pcode>
</callfixup>
```

<callotherfixup>

Table 6.

Attributes and Children	
`targetop`	Name of the *CALLOTHER* operator to inject.
`<pcode>`	Description of p-code to inject.

<pcode>

Table 7.

Attributes and Children		
`<input>`		(0 or more) Description of formal input parameter.
	`name`	Name of the specific input symbol.
	`size`	Expected size of the parameter in bytes.
`<output>`		(0 or more) Description of formal output parameter.
	`name`	Name of the specific output symbol.
	`size`	Expected size of output in bytes.
`<body>`		P-code to inject.
	`text`	

The `<callotherfixup>` is similar to a `<callfixup>` tag but is used to describe injections that replace user-defined p-code operations, rather than `CALL` operations. User-defined p-code operations, referred to generically as `CALLOTHER` operations, are *black-box* operations that a SLEIGH specification can define to encapsulate complicated (or esoteric) actions performed by the processor. The specification must define a unique name for each such operation. The `targetop` attribute links the p-code described here to the specific operation via this name.

As with any p-code operation, the `CALLOTHER` takes formal varnodes as inputs and/or outputs. These varnodes can be referred to in the injection `<body>` by predefining them using `<input>` or `<output>` tags. The sequence of `<input>` tags correspond in order to the input parameters of the `CALLOTHER`, and a `<output>` tag corresponds to output varnode if present. The tags listed here **must** match the number of input and output parameters in the actual p-code operation, or an exception will be thrown during p-code generation. The optional `size` attribute in each tag will, if present, impose a size restriction on the parameter as well.

As with a `<callfixup>`, the `<body>` tag is fed straight to the SLEIGH semantic parser. It can refer to registers via their symbolic name defined in SLEIGH, it can refer to the operator parameters via their `<input>` or `<output>` names, and it can also refer to `inst_start` and `inst_next` as addresses describing the instruction containing the `CALLOTHER`.

Example 6.

```
<callotherfixup targetop="saturate">
  <pcode>
    <input name="in1" size="4"/>
    <input name="in2" size="4"/>
    <body><![CDATA[
      in1 = in1 + in2;
      if (in1 < 0x10000) goto <end>;
      in1 = 0xffff;
      <end>
    ]]></body>
  </pcode>
</callotherfixup>
```

<prefersplit>

Table 8.

Attributes and Children	
style	Strategy for splitting: *inhalf*
<register> or <varnode>	(1 or more) *varnode* tags

This tag is designed to mark specific registers as *packed*, containing multiple logical values that need to be split. The decompiler attempts to split up any operator that reads or writes the register into multiple p-code operations that operate on each logical value individually.

The tag lists one or more **varnode tags** describing the registers or other storage locations that need to be split. The *style* attribute indicates how the storage locations should be split. Currently the only accepted style value is "inhalf", which means that each varnode should be split into two equal pieces.

Splitting a varnode is only possible if the all p-code operations it is involved in don't mix their action across the logical pieces. If this is not possible, the p-code will not be altered for that particular varnode.

Example 7.

```
<prefersplit style="inhalf">
  <register name="xr0"/>
  <register name="xr1"/>
  <register name="xr2"/>
<prefersplit>
```

<aggressivetrim>

Table 9.

Attributes and Children	
signext	(Optional) *true* if sign-extension should be aggressively trimmed

This tag tells the decompiler that p-code extension operations are likely to be a side-effect of the processor and are obscuring what is just the manipulation of the smaller logical value. The decompiler normally trims extensions and other operations where it can prove that the most significant bytes of the result are unused. This tag lets the decompiler be more aggressive when use of the extended bytes is more indeterminate. It can assume that extensions into sub-function parameters and into the return value are extraneous.

The *signext* attribute turns the behavior on specifically for the sign-extension operation. Currently there is no toggle for zero-extensions.

Example 8.

```
<aggressivetrim signext="true"/>
```

Compiler Datatype Organization

<data_organization>

Table 10.

Attributes and Children		
`<absolute_max_alignment>`		(Optional) Maximum alignment possible across all datatypes (0 indicates no maximum)
	`value`	
`<machine_alignment>`		(Optional) Maximum useful alignment for the underlying architecture
	`value`	
`<default_alignment>`		(Optional) Default alignment for any datatype that isn't structure, union, array, or pointer and whose size isn't in the size/alignment map
	`value`	

Attributes and Children		
`<default_pointer_alignment>`		(Optional) Default alignment for a pointer that doesn't have a size
	`value`	
`<pointer_size>`		(Optional) Size of a pointer
	`value`	
`<pointer_shift>`		(Optional) Left-shift amount, in bits, for shifted pointer datatypes
	`value`	
`<wchar_size>`		(Optional) Size of "wchar", the wide character datatype
	`value`	
`<short_size>`		(Optional) Size of "short" and other short integer datatypes
	`value`	
`<integer_size>`		(Optional) Size of "int" and other integer datatypes
	`value`	
`<long_size>`		(Optional) Size of "long" and other long integer datatypes
	`value`	
`<long_long_size>`		(Optional) Size of "longlong" integer datatypes
	`value`	
`<float_size>`		(Optional) Size of "float" and other floating-point datatypes
	`value`	
`<double_size>`		(Optional) Size of "double" and other double precision floating-point datatypes
	`value`	
`<long_double_size>`		(Optional) Size of "longdouble" floating-point datatypes
	`value`	
`<size_alignment_map>`		(Optional) Size to alignment map

The `<data_organization>` tag provides information about the sizes of core datatypes and how the compiler typically aligns datatypes. These are required so analysis can determine the proper in-memory layout of datatypes, such as those described by C/C++ style header files. Both sizes and alignments are specified in bytes by using the integer `value` attribute in the corresponding tag. An alignment value indicates that the compiler chooses a byte address that is a multiple of that value as the start of that datatype. A value of 1 indicates *no alignment*. Most atomic datatypes get their alignment information from the `<size_alignment_map>`. If the size of a particular datatype isn't listed in the map, the `<default_alignment>` value will be used.

<size_alignment_map>

Table 11.

Attributes and Children	
`<entry>`	(0 or more) Alignment information for a particular size
`size`	Size of datatype in bytes
`alignment`	The alignment value

Each `<entry>` maps a specific size to a specific alignment. Ghidra satisfies requests for the alignment of all atomic datatypes (except pointers) by consulting this map. If it doesn't contain the particular size, Ghidra reverts to the `<default_alignment>` subtag in the parent `<data_organization>` tag. Its typical to only provide alignments for sizes which are a power of 2.

Example 9.

```
<data_organization>
   <absolute_max_alignment value="0" />
   <machine_alignment value="2" />
   <default_alignment value="1" />
   <default_pointer_alignment value="4" />
   <pointer_size value="4" />
   <wchar_size value="4" />
   <short_size value="2" />
   <integer_size value="4" />
   <long_size value="4" />
   <long_long_size value="8" />
   <float_size value="4" />
   <double_size value="8" />
   <long_double_size value="12" />
   <size_alignment_map>
        <entry size="1" alignment="1" />
        <entry size="2" alignment="2" />
        <entry size="4" alignment="4" />
        <entry size="8" alignment="4" />
   </size_alignment_map>
</data_organization>
```

<enum>

Table 12.

Attributes and Children	
`size`	Default size of an enumerated datatype
`signed`	(Optional) *true* or *false* : Is an enumeration viewed as a signed integer

This is a **deprecated** tag.

<funcptr>

Table 13.

Attributes and Children	
`align`	Number of alignment bytes for functions

Some compilers rely on the alignment of code addresses to provide extra bits of space in function pointers where extra internal information can be stored. On ARM chips in particular, the processor itself supports an ARM/THUMB transition bit in code addresses, which are always at least 2 byte aligned. This tag informs the decompiler of this region of encoding in function pointers so that it can filter it out, allowing it to find the correct address in various situations. The `align` attribute should always be a power of 2 corresponding to the number of bits a compiler might use for additional storage.

Example 10.

```
<funcptr align="2"/>
```

Compiler Scoping and Memory Access

<global>

Table 14.

Attributes and Children		
`<register>`		(0 or more) Specific register to be marked as global
	`name`	Name of register
`<range>`		(0 or more) Range of addresses to be marked as global
	`space`	Address space of the global region
	`first`	(Optional) Starting offset of the region
	`last`	(Optional) Ending offset of the region

The `<global>` tag marks specific memory regions as storage locations for the compiler's global variables. The word *global* here refers to the standard scoping concept for variables in high-level source code, meaning that the variable or memory location is being used as permanent interfunction storage. This tag informs the decompiler's *discovery* of the scope of particular memory locations. Any location not marked as global in this way is assumed to be local/temporary storage.

Example 11.

```
<global>
  <range space="ram"/>
</global>
```

<readonly>

Table 15.

Attributes and Children		
`<register>`		(0 or more) Specific register to be marked as read-only
	`name`	Name of register
`<range>`		(0 or more) Range of addresses to be marked as read-only
	`space`	Address space of the read-only region
	`first`	(Optional) Starting offset of the region
	`last`	(Optional) Ending offset of the region

The `<readonly>` tag labels a specific region as read-only. From the point of view of the compiler, these memory locations hold constant values. This allows the decompiler to propagate these constants and potentially perform additional simplification. This tag is not very common because most read-only memory sections are determined dynamically from the executable header.

Example 12.

```
<readonly>
  <range space="ram" first="0x3000" last="0x3fff"/>
</readonly>
```

<nohighptr>

Table 16.

Attributes and Children		
`<register>`		(0 or more) Specific register to be marked as not addressable
	`name`	Name of register
`<range>`		(0 or more) Range of addresses to be marked as not addressable
	`space`	Address space of the unaddressable region
	`first`	(Optional) Starting offset of the region
	`last`	(Optional) Ending offset of the region

The `<nohighptr>` tag describes a memory region into which the compiler does not expect to see pointers from any high-level source code. This is slightly different from saying that there are absolutely no indirect references into the region. This tag is really intended to partly address the modeling of *memory-mapped registers*. If a common register is addressable through main memory, this can confound decompiler analysis because even basic simplifications are blocked by writes through dynamic pointers that might affect the register. This tag provides an apriori guarantee that this is not possible for the marked registers.

Example 13.

```
<nohighptr>
  <range space="DATA" first="0xf80" last="0xfff"/>
</nohighptr>
```

Compiler Special Purpose Registers

<stackpointer>

Table 17.

Attributes and Children	
`register`	Name of register to use as stack pointer
`space`	Address space that will hold the *stack*
`growth`	(Optional) *negative* or *positive*
`reversejus-tify`	(Optional) *true* or *false*

The `<stackpointer>` tag informs Ghidra of the main stack mechanism for the compiler. The `reg-ister` attribute gives the name of the register that holds the current offset into the stack, and the `space` attribute specifies the name of the address space that holds the actual data. This tag triggers the creation of a formal *stack* space. A separate stack space exists virtually for each function being analyzed where offsets are calculated relative to the incoming value of this register. This provides a *concrete* stor-age location for a function's local variables even though the true location is dynamically determined.

By default the stack is assumed to grow in the *negative* direction, meaning that entries which are deeper on the stack are stored at larger offsets, and each new entry pushed on the stack causes the stackpointer register to be decremented. But this can be changed by setting the `growth` attribute to *positive*, which reverses the direction that new entries are pushed on the stack.

<returnaddress>

Table 18.

Attributes and Children	
<register> or <varnode>	One *varnode* tag

This tag describes how the return address is stored, upon entry to a function. It takes a single varnode sub-tag describing the storage location (See the section called “Varnode Tags”). In many cases, the de-compiler can eliminate return value data-flow without knowing this information because the value is never used within the function and other parameter passing is explicitly laid out. Sometimes however, return values can look like part of a structure allocated on the stack or can be confused with other da-ta-flow. In these cases, the `<returnaddress>` tag can help by making the standard storage location explicit.

The storage location of the return address is actually a property of a prototype model. This tag defines a global default for all prototype models, but it can be overridden for individual prototype models. See the section called “<returnaddress>”.

Example 14.

```
<returnaddress>
  <varnode space="stack" offset="0" size="4"/>
</returnaddress>
```

Parameter Passing

A *prototype model*, in Ghidra, is a set of rules for determining how parameters and return values are passed between a function and its subfunction. For a high-level language (such as C or Java), a function prototype is the ordered list of parameters (each specified as a name and a datatype) that are passed to the function as input plus the optional value (specified as just a dataype) returned by the function. A prototype model specifies how a compiler decides which storage locations are used to hold the actual values at run time.

From a reverse engineering perspective, Ghidra also needs to solve the inverse problem: given a set of storage locations (registers and stack locations) that look like they are inputs and outputs to a function, determine a high-level function prototype that produces those locations when compiled. The same prototype model is used to solve this problem as well, but in this case, the solution may not be unique, or can only be exactly derived from information that Ghidra doesn't have.

Describing Parameters and Allocation Strategies

The `<prototype>` tag encodes details about a specific prototype model, within a compiler specification. A given compiler spec can have multiple prototype models, which are all distinguished by the mandatory *name* attribute for the tag. Other Ghidra tools refer to prototype model's by this name, and it must be unique across all models in the compiler spec. All `<prototype>` tags must include the subtags, `<input>` and `<output>`, which list storage locations (registers, stack, and other varnodes) as the raw material for the prototype model to decide where parameters are stored for passing between functions. The `<input>` tag holds the resources used to pass input parameters, and `<output>` describes resources for return value storage. A resource is described by the `<pentry>` tag, which comes in two flavors. Most `<pentry>` tags describe a storage location to be used by a single variable. If the tag has an *align* attribute however, multiple variables can be allocated from the same resource, where different variables must be aligned relative to the start of the resource as specified by the attribute's value.

How `<pentry>` resources are used is determined by the prototype model's *strategy*. This is specified as an optional attribute to the main `<prototype>` tag. There are currently only two strategies: *standard* and *register*. If the attribute is not present, the prototype model defaults to the *standard* strategy.

Standard Strategy

For this strategy, the `<pentry>` subtags under the `<input>` tag are viewed as an ordered resource list. When assigning storage locations from a list of datatypes, each datatype is evaluated in order. The first `<pentry>` from the resource list that fits the datatype and hasn't been fully used by previous datatypes is assigned to that datatype. In this case, the `<input>` tag lists varnodes in the order that a compiler would dole them out when given a list of parameters to pass. Integer or pointer values are usually passed first in specially designated registers rather than on the stack if there are not enough available registers. There can one stack-based `<pentry>` at the end of the list that will typically match any number of parameters of any size or type.

If there are separate `<pentry>` tags for dedicated floating-point registers, the standard strategy treats them as a separate resource list, independent of the one for integer and pointer datatypes. The `<pen-`

`try>` tags specifying floating-point registers are listed in the same `<input>` tag, immediately after the integer registers, and are distinguished by the `metatype="float"` attribute labeling the individual tags.

For the inverse case, where the decompiler must infer a prototype from data-flow and liveness, the standard strategy expects there to be no **gaps** in the usage of the (either) resource list. For a putative input varnode to be considered a formal parameter, it must occur somewhere in the `<pentry>` resource list. If there is a gap, i.e. the second `<pentry>` occurs as a varnode but not the first, then the decompiler will fill in the gap by creating an extra *unused* parameter. Or if the gap is too big, the original input varnode will not be considered a formal parameter.

Register Strategy

This allocation strategy is designed for software with a lot of hand-coded assembly routines that are not sticking to a particular parameter passing strategy. The idea is to provide `<pentry>` tags for any register that might conceivably be considered an input location. Then the input varnodes for a function that have a corresponding `<pentry>` are automatically promoted to formal parameters. In practical terms, this strategy behaves in the same way as the Standard strategy, except that in the reverse case, the decompiler does not care about gaps in the resource list. It will not fill in gaps, and it will not throw out putative inputs because of large gaps.

When assigning storage locations from a list of datatypes, the same algorithm is applied as in the standard strategy. The first `<pentry>` that hasn't been used and that fits the datatype is assigned. Note that this may not make as much sense for hand-coded assembly.

<default_proto>

Table 19.

Attributes and Children	
`<prototype>`	Specification for the default prototype

There must be exactly one `<default_proto>` tag, which contains exactly one `<prototype>` subtag. Other `<prototype>` tags can be listed outside of this tag. The designated default prototype model. Where users are given the option of choosing from among different prototype models, the name "default" is always presented as an option and refers to this prototype model. It is also used in some situations where the prototype model is unknown but analysis needs to proceed.

<prototype>

Table 20.

Attributes and Children	
`name`	The name of the prototype model
`extrapop`	Amount stack pointer changes across a call or *unknown*
`stackshift`	Amount stack changes due to the call mechanism
`type`	(Optional) Generic calling convention type: *stdcall*, *cdecl*, *fastcall*, or *thiscall*
`strategy`	(Optional) Allocation strategy: *standard* or *register*

Attributes and Children		
`<input>`		Resources for input variables
	`pointermax`	(Optional) Max size of parameter before converting to pointer
	`thisbeforeretpointer`	(Optional) *true* if *this* pointer comes before hidden return pointer
	`killedbycall`	(Optional) *true* indicates all input storage locations are considered killed by call
	`<pentry>`	(1 or more) Storage resources
`<output>`		Resources for return value
	`killedbycall`	(Optional) *true* indicates all output storage locations are considered killed by call
	`<pentry>`	(1 or more) Storage resources
`<returnaddress>`		(Optional) Storage location of return value
`<unaffected>`		(Optional) Registers whose value is unaffected across calls
`<killedbycall>`		(Optional) Registers whose value does not persist across calls
`<likelytrash>`		(Optional) Registers that may hold a trash value entering the function
`<localrange>`		(Optional) Range of stack locations that may hold mapped local variables

The `<prototype>` tag specifies a prototype model. It must have a *name* attribute, which gives the name that can be used both in the Ghidra GUI and at other points within the compiler spec. The *strategy* attribute indicates the allocation strategy, as described below. If omitted the strategy defaults to *standard*.

Every `<prototype>` must specify the *extrapop* attribute. This indicates the change in the stack pointer to expect across a call, within the p-code model. For architectures where a call instruction pushes a return value on the stack, this value will usually be positive and match the size of the stack-pointer in bytes, indicating that a called function usually pops the return value itself and changes the stack pointer in a way not apparent in the (callers) p-code. For architectures that use a link register to store the return address, *extrapop* is usually zero, indicating to the decompiler that it can expect the stack pointer value not to change across a call. The attribute can also be specified as *unknown*. This turns on the fairly onerous analysis associated with the Microsoft *stdcall* calling convention, where functions, upon return, pop off their own stack parameters in addition to the return address.

The *stackshift* attribute is also mandatory and indicates the amount the stack pointer changes just due to the call mechanism used to access a function with this prototype. The call instruction for many processors pushes the return address onto the stack. The *stackshift* attribute would typically be 2, 4, or 8, matching the code address size, in this case. For link register mechanisms, this attribute is set to zero.

The *type* attribute can be used to associate one of Ghidra's *generic calling convention* types with the prototype. The possible values are: *stdcall*, *cdecl*, *fastcall*, and *thiscall*. Each of these values can be assigned to at most one calling convention across the compiler specification. Generic calling conventions are used to encode calling convention information in a Ghidra datatype, like a FunctionDefinitionDataType, which can apply to more than one program or architecture.

<input>

The `<input>` tag lists the resources used to pass input parameters to a function with this prototype. The varnodes used for passing are selected by an *allocation strategy* (See the section called “Describing Parameters and Allocation Strategies”) from among the resources specified here. The `<input>` tag contains a list of `<pentry>` sub-tags describing the varnodes. Depending on the allocation strategy, the ordering is typically important.

The *killedbycall* attribute if true indicates that all storage locations listed in the `<input>` should be considered as killed by call (See the section called “<killedbycall>”). This attribute is optional and defaults to false.

The *pointermax* attribute can be used if there is an absolute limit on the size of datatypes passed directly using the standard resources. If present and non-zero, the attribute indicates the largest number of bytes for a parameter. Bigger inputs are assumed to have a pointer passed instead. When a user specifies a function prototype with a big parameter, Ghidra will automatically allocate a storage location that holds the pointer. By default, this substitution does not occur, and large parameters go through the normal resource allocation process and are assigned storage that holds the whole value directly.

The *thisbeforeretpointer* indicates how the two hidden parameters, the *this* pointer and the hidden return pointer, are ordered on the stack, in the rare case where both occur in a single prototype. If the attribute is true, the *this* pointer comes first. By default, the hidden return will come first.

The following is an example tag using the standard allocation strategy with 3 integer registers and 2 floating-point registers. If there are more parameters of either type, the compiler allocates storage from the stack.

Example 15.

```
<input>
  <pentry minsize="1" maxsize="8" metatype="float">
    <register name="f1"/>
  </pentry>
  <pentry minsize="1" maxsize="8" metatype="float">
    <register name="f2"/>
  </pentry>
  <pentry minsize="1" maxsize="4">
    <register name="a0"/>
  </pentry>
  <pentry minsize="1" maxsize="4">
    <register name="a1"/>
  </pentry>
  <pentry minsize="1" maxsize="4">
    <register name="a2"/>
  </pentry>
  <pentry minsize="1" maxsize="500" align="4">
    <addr offset="16" space="stack"/>
  </pentry>
</input>
```

<output>

The handling of `<pentry>` subtags within the `<output>` tag is slightly different than for the input case. Technically, this tag is sensitive to the *allocation strategy* selected for the prototype. Currently however, all (both) strategies behave the same for the output parameter.

When assigning a storage location for a return value of a given data-type, the first `<pentry>` within list that matches the data-type is used as the storage location. If none of the `<pentry>` storage locations fit the data-type, a *Hidden Return Parameter* is triggered. An extra hidden input parameter is passed which holds a pointer to where the function will store the return value.

In the inverse case, the decompiler examines all (possible) output varnodes that have a corresponding `<pentry>` tag in the resource list. The varnode whose corresponding tag occurs the earliest in the list becomes the formal return value for the function. If an output varnode matches no `<pentry>`, then it is rejected as a formal return value.

Example 16.

```
<output killedbycall="true">
  <pentry minsize="4" maxsize="10" metatype="float" extension="float">
    <register name="ST0"/>
  </pentry>
  <pentry minsize="1" maxsize="4">
    <register name="EAX"/>
  </pentry>
  <pentry minsize="5" maxsize="8">
    <addr space="join" piece1="EDX" piece2="EAX"/>
  </pentry>
</output>
```

<pentry>

Table 21.

Attributes and Children		
`minsize`		Size (in bytes) of smallest variable stored here
`maxsize`		Size (in bytes) of largest variable stored here
`align`		(Optional) Alignment of successive locations within this entry
`metatype`		(Optional) Restriction on datatype: *unknown*, *float*, *int*, *uint*, or *ptr*
`extension`		(Optional) How small values are extended: *sign*, *zero*, *inttype*, *float*, or *none*
`<register>`		Storage location of the entry
	`name`	Name of register
`<addr>`		(alternate form)
	`space`	Address space of the location
	`offset`	Offset (in bytes) of location

The `<pentry>` tag describes the individual memory resources that make up both the `<input>` and `<output>` resource lists. These are consumed by the allocation strategy as it assigns storage for parameters and return values. Attributes describe restrictions on how a particular `<pentry>` resource can be used.

The storage for the entry is specified by either the `<register>` or the `<addr>` subtag. The `minsize` and `maxsize` attributes restrict the size of the parameter to which the entry is assigned, and the `metatype` attribute restricts the type of the parameter.

Metatype refers to the *class* of the datatype, independent of size: integer, unsigned integer, floating-point, or pointer. The default is `unknown` or no type restriction. The `<metatype>` can be used to split out a separate floating-point resource list for some allocation strategies. In the *standard* strategy for instance, any `<pentry>` that has the attribute `metatype="float"` is pulled out into a separate list from all the other entries.

The optional `extension` attribute indicates that variables are extended to fill the entire location, if the datatype would otherwise occupy fewer bytes. The *type* of extension depends on this attribute's value: `zero` for zero extension, `sign` for sign extension, and `float` for floating-point extension. A value of `inttype` indicates the value is either sign or zero extended depending on the original datatype. The default is `none` for no extension.

The `align` attribute indicates that multiple variables can be drawn from the `pentry` resource. The first variable occupies bytes starting with the address of the storage location specified in the tag. Additional variables start at the next available aligned byte. The attribute value must be a positive integer that specifies the alignment. This is typically used to model parameters pulled from a stack resource. The example below draws up to 500 bytes of parameters from the stack, which are 4 byte aligned, starting at an offset of 16 bytes from the initial value of the stack pointer.

Example 17.

```
<pentry minsize="1" maxsize="500" align="4">
  <addr space="stack" offset="16"/>
</pentry>
```

<returnaddress>

Table 22.

Attributes and Children	
<register> or <varnode>	One *varnode* tag

This is an optional tag that describes where the *return address* is stored, upon entering a function. If present, it overrides the default value for functions that use this particular prototype model. (See the section called “<returnaddress>”) It takes a single **varnode tag** describing the storage location.

Example 18.

```
<returnaddress>
  <register name="RA" />
</returnaddress>
```

<unaffected>

Table 23.

Attributes and Children	
<register> or <varnode>	(1 or more) *varnode* tags

This tag lists one or more storage locations that the compiler knows will not be modified by any sub-function. Each storage location is specified as a **varnode tag**.

By contract, sub-functions must either not touch these locations at all, or they must save off the value and then restore it before returning to their caller. Many ABI documents refer to these as *saved registers*. Fundamentally, this allows the decompiler to propagate values across function calls. Without this tag, because it is generally looking at a single function in isolation, the decompiler doesn't have enough information to safely allow this kind of propagation.

Example 19.

```
<unaffected>
  <register name="ESP"/>
  <register name="EBP"/>
</unaffected>
```

<killedbycall>

Table 24.

Attributes and Children	
<register> or <varnode>	(1 or more) *varnode* tags

This tag lists one or more storage locations, each specified as a **varnode tag**, whose value should be considered killed by call.

A register or other storage location is *killed by call* if, from the point of view of the calling function, the value of the register before a sub-function call is unrelated to its value after the call. This is effectively the opposite of the `<unaffected>` tag which specifies that the value is unchanged across the call.

A storage location marked neither `<unaffected>` or `<killedbycall>` is treated as if it *may* hold different values before and after the call. In other words, the storage location represents the same high-level variable before and after, but the call may modify the value.

Example 20.

```
<killedbycall>
  <register name="ECX"/>
  <register name="EDX"/>
</killedbycall>
```

<likelytrash>

Table 25.

Attributes and Children	
<register> or <varnode>	(1 or more) *varnode* tags

This tag lists one or more storage locations specified as a **varnode tag**. In specialized cases, compilers can move around what seem like input values to functions, but the values are actually unused and the movement is incidental. The canonical example, is the push of a register on the stack, where the code is simply trying to make space on the stack.

If there is movement and no other explicit manipulation of the input value in a storage location tagged this way, the decompiler will treat the movement as dead code.

Example 21.

```
<likelytrash>
  <register name="ECX"/>
</likelytrash>
```

<localrange>

Table 26.

Attributes and Children		
`<range>`		(1 or more) Range of bytes eligible for local variables
	`space`	Address space containing range (Usually "stack")
	`first`	(Optional) Starting byte offset of range, default is 0
	`last`	(Optional) Ending byte offset, default is maximal offset of space

This tag lists one or more `<range>` tags that explicitly describe all the possible ranges on the stack that can hold mapped local variables other than parameters. Individual functions will be assumed to use some subset of this region. The *first* and *last* attributes to the `<range>` tag give offsets relative to the incoming value of the stack pointer. This affects the decompiler's reconstruction of the stack frame for a function and parameter recovery.

Omitting this tag and accepting the default is often sufficient. The default sets the local range as all bytes not yet pushed on the stack, where the incoming stack pointer points to the last byte pushed. An explicit tag is useful when a specific region needs to be added to or excised from the default. The following example is for the 64-bit x86 prototype model, where the caller reserves extra space on the stack for register parameters that needs to be added to the default. The `<localrange>` tag replaces the default, so it needs to specify the default range if it wants to keep it.

Example 22.

```
<localrange>
  <range space="stack" first="0xfffffffffff0bdc1" last="0xffffffffffffffff"/>
  <range space="stack" first="8" last="39"/>
</localrange>
```