INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Devil Language

Laurent Réveillère, Fabrice Mérillon, Charles Consel, Renaud Marlet, Gilles Muller

N°0244

Octobre 2000

THÈME 2

Rapport technique

ISSN 0249-0803

The Devil Language

Laurent Réveillère, Fabrice Mérillon, Charles Consel, Renaud Marlet, Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport technique n°0244 — Octobre 2000 — ?? pages

Abstract: Devil is a Domain-specific language dedicated to defining the basic communication with a device. Unlike a general-purpose language, Devil allows a specification to be checked for consistency. This not only improves the safety of the interaction with the device but also uncovers bugs early in the development process. A compiler automatically generates from a Devil specification efficient low-level code to communicate with the device. This report presents the reference manual of the Devil language.

Key-words: Devil, Domain Specific Language, Devices

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Le langage Devil

Résumé : Devil est un langage dédié pour définir les communications de base avec un périphérique. Contrairement à un langage généraliste, Devil permet de vérifier la consistance d'une spécification. Cela ne fait pas que renforcer la sûreté des interactions avec le périphérique mais aussi permet de découvrir des bogue plus tôt dans le processus de développement. Un compilateur génère automatiquement à partir d'une spécification Devil un code bas niveau efficace pour communiquer avec le périphérique. Ce rapport présente le manuel de référence du langage Devil.

Mots-clé : Devil, Langage dédié, périphériques

Foreword

This document is intended as a reference manual for the Devil language. It lists the language constructs, gives their precise syntaxs and an informal semantics. It is by no means a tutorial introduction to the language.

Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in a typewriter font (`like this`). Non-terminals are set in an italic font (*like that*). The vertical bar | denotes an alternative in a rule. Parentheses $(\ldots)$ denote grouping. Parentheses with a trailing star sign $(\ldots)^*$ denote zero, one or several occurrences of the enclosed item. Parentheses with a trailing plus sign $(\ldots)^+$ denote one or several occurrences of the enclosed item. Parentheses with a trailing question mark sign $(\ldots)^?$ denote an optional item.

Availability

The complete Devil distribution (including the taz compiler) is freely available on the World Wide Web, `http://www.irisa.fr/compose/devil`

1 Introduction

Devil is a domain-specific language for specifying the functional interface of a device. Concretely, a device can be described by three layers of abstractions: *ports*, *registers*, and *device variables*. The entry point of a Devil specification is the declaration of a device, parameterized by ports or range of ports, which abstract physical addresses. Ports then allow device registers to be declared; these define the granularity of interaction with the device. Finally, device variables are defined from registers, forming the functional interface to the device.

Only device variables are visible from outside a Devil description; ports and registers are hidden since these abstractions are not part of the functional interface of the device. In fact, the Devil compiler (named Taz) generates for each variable two C procedures which permit to write or read the variable by emitting the proper I/O operations.

2 Lexical conventions

Blanks

The following characters are considered as blanks : space, newline and horizontal tabulation. Blanks separate adjacent identifiers, literals and keywords that would be otherwise confused as a single identifier, literal or keyword. Apart from that, they are ignored.

Comments

Comments are C-like comments. They start with the two characters `/*` and end with the characters `*/`. C++ comments style can also be used; all characters from the two characters `//` till the end of the line are considered as comments too. Comments are treated as blanks.

Identifiers

Identifiers are a sequence of letters, digits and `_` (the underscore character) starting with a letter. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of an identifier.

$$\begin{array}{lll} \textit{ident} & ::= & \textit{letter}\ (\textit{letter} \mid \textit{digit} \mid \texttt{\_})^* \\ \textit{letter} & ::= & \texttt{A..Z} \mid \texttt{a..z} \\ \textit{digit} & ::= & \texttt{0..9} \end{array}$$

Integer literals

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. By default, integers literals are in decimal (radix 10).

$$\begin{array}{lll} \textit{integer-literal} & ::= & (\texttt{0..9})^+ \\ & \mid & \texttt{0x}\ (\texttt{0..9} \mid \texttt{A..F} \mid \texttt{a..f})^+ \\ & \mid & \texttt{0o}\ (\texttt{0..7})^+ \end{array}$$

The following prefixes select a different radix:

Prefix	Radix
0x	hexadecimal (radix 16)
0o	octal (radix 8)

Note that the initial 0 is digit zero and the o for octal is letter o.

Boolean literals

The *boolean* type has two possible values, represented by the literals `true` and `false`. A boolean literal is always of type boolean.

$$\begin{array}{lll} \textit{boolean-literal} & ::= & \texttt{true} \\ & \mid & \texttt{false} \end{array}$$

Bit literals

Bit literals are delimited by `'` (single quote) characters.

$$\begin{array}{lll} \textit{bits-literal} & ::= & \texttt{'}\ (\texttt{bit})^+\ \texttt{'} \\ \textit{bit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{*} \mid \texttt{.} \end{array}$$

Prefix and infix operators

The following tokens are the Devil operators :

```
@    ..   .   =>   <=   <=>
=    #
+    -    *   /
==   !=   <   >    >=
&&   ||   !
```

Note that sequences of “operator characters”, such as `!=` or `<=>` are read as a single token.

Keywords

The identifiers below are reserved keywords:

```
as          bit       bool       const    device  else
except      false     for        if       int     mask
port        post      pre        private  read    register
serialized  set       structure  trigger  true    type
unused      variable  volatile   write
```

The following character sequences are also reserved:

```
{  }  [  ]  (  )
:  ,  =  ;
```

Ambiguities

Lexical ambiguities are resolved according to the “longest match” rule: when a character sequence can be decomposed into tokens in several different ways, the resulting decomposition is the one with the longest first token.

3 Device Specification

device-spec	::=	`device` *ident* `(`*device-params*`)` `{` (*object-def* `;`)$^{+}$ `}`
device-params	::=	*device-param* (`,` *device-param*)*
device-param	::=	*ident* `:` *type-expr* `port` (`@` `{` *integer-ranges* `}`)$^{?}$
integer-ranges	::=	*integer-range* (`,` *integer-range*)*
integer-range	::=	*integer-literal*
	|	*integer-literal* `..` *integer-literal*
object-def	::=	*objdef*
	|	`if` `(`*expr*`)` *objdef-body* (`else` *objdef-body*)$^{?}$
objdef-body	::=	*objdef*
	|	`{` (*objdef* `;`)$^{+}$ `}`
objdef	::=	*register-definition*
	|	*structure-definition*
	|	*type-definition*
	|	*variable-definition*
	|	*unused-declaration*

A device specification is introduced by the `device` keyword, and consists of an identifier, followed by a comma separated list of parameters, and then one or more Devil object definitions . Devil objects are register, structure, type or variable. The identifier is the name of the device being defined.

Device parameters

A device parameter named *base* specifies the number of bits that can be exchanged at the *base* address. The optional construct `@ {` *integer-ranges* `}` allows one to define a set of integer offsets that can be added to the *base* address in order to compute an other address with the same characteristics. Note that the device parameter definition

base `: bit[8] port`

is equivalent to

base `: bit[8] port @ {0}`

Conditional definitions

The expression `if (`*expr*`)` *def-body*$_1$ `else` *def-body*$_2$ allows one to conditionally define a Devil object. A Devil object cannot be re-defined; the same identifier cannot be used for several differents Devil objects, even if its definition is conditional. A definition in *def-body*$_1$ (respectively *def-body*$_2$) can be accessed at a time t if and only if the expression *expr* can be evaluated to `true` (respectively `false`) in the environment present at time t. The `else` *def-body*$_2$ part can be omitted, in which case it defaults to an empty definition.

Expressions of conditional definitions

The expression of a conditional definition is an expression where identifiers can refer to device parameters or variables. The expression is evaluated each time an access to a variable conditionally defined is performed. The identifier name of a read variable is evaluated to the previous value read. The identifier name of a write or read-write variable is evaluated to the previous value assigned to this variable. Moreover, code can be inserted in the generated interface in order to check if a variable has been previously read or written.

4 Expressions

```
expr             ::=  boolean-literal
                  |   integer-literal
                  |   ident
                  |   ( expr )
                  |   uop expr
                  |   expr binary-operator expr
unary-operator   ::=  - | !
binary-operator  ::=  + | - | * | / | %
                  |   && | ||
                  |   == | != | < | > | <= | >=
```

The table below shows the relative precedence and associativity of operators. The operators are ordered top-down from the highest to the lowest precedence.

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>!</td><td>right to left</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ -</td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>== !=</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
</table>

Note that the unary operator `-` has higher precedence than binary operators.

5 Registers

This section describes the register declaration aspects of a Devil specification.

```
register-definition  ::=  register ident ((register-params))? = register-def
register-params      ::=  register-param (, register-param)*
register-param       ::=  ident : type-expr
register-def         ::=  (register-ports | reg-params) (, register-attribute)* : type-expr
register-ports       ::=  register-port (, register-port)?
reg-params           ::=  reg-param (, reg-param)?
register-port        ::=  (rw)? regport-def
exprs                ::=  expr (, expr)*
regport-def          ::=  ident
                      |   ident @ expr
reg-param            ::=  (rw)? ident (exprs)
register-attribute   ::=  (rw)? pre { (action-definition ;)+ }
                      |   (rw)? post { (action-definition ;)+ }
                      |   (rw)? mask bits-litteral
rw                   ::=  read
                      |   write
action-definition    ::=  ident = expr
                      |   ident.ident = expr
                      |   ident[expr] = expr
                      |   ident = { (action-definition ;)+ }
```

A register identifies data that are present in the device as well as how these data can be accessed (read or written). If a register definition only has a read (resp. write) part, the register is said to be a read-only (resp. a write-only) register. Otherwise it is considered as a read-write register. The type of a register consists of a number of bits and is the same for both the read and write parts. In addition, the register definition may contain the description of other four components: port, bit-mask, pre-actions and post-actions.

Ports

A port is defined by the expression *ident* `@` *expr* which can be optionally preceded by a read or write modifier. *ident* is the identifier name of a device parameter. *expr* is a static expression that must be evaluable at compile time. The `@` *expr* can be omitted, in which case it defaults to `@ 0`. One or two ports must be provided in a register definition. The read (resp. write) part of a register is defined if and only if a read (resp. write) port definition is provided. If only one port is provided without a modifier, its definition is used both for the read and write parts.

```
device logitech_busmouse (base : bit[8] port @ {0..3})
{ ... }
```

Bit-masks

A bit-mask is introduced by the `mask` keyword, optionally preceded by a read or write modifier and followed by a *bits-literal*. If a bit-mask is preceded by a read modifier (respectively write modifier), its definition applies only to the read part (respectively write part) of the register definition. The *bits-literal* of a bit-mask is a sequence of n mask elements (`0`,`1`, `*` or `.`) where n stands for the size (number of bits) of the register. Mask elements are numbered from right to left so that the right hand side element of a bit-mask masks the lowest bit of the corresponding register. The table below shows, for each mask element, the constraint associated to its corresponding bit.

mask	bit mapped	bit value
`0`	no	0
`1`	no	1
`*`	no	any
`.`	yes	bit variable value

If no bit-mask is provided, it defaults to $\texttt{'}\underbrace{\texttt{......}}_{n}\texttt{'}$ where n stands for the size of the register.

Unused declarations

The bit of a register masked with . (dot sign) has to be mapped by a device variable. The unused declaration can be used when the variable is conditionally defined to declare bits that are conditionnaly mapped.

unused-declaration ::= `unused bit` *register-bits*

The unused declaration is introduced by the `unused bit` keywords, optionally preceded by a read or write modifier, and followed by a *register-bits* expression (see section **??**). An unused declaration has to be conditionally defined since a bit which is always irrelevant must be masked with the `*` sign.

Pre and post actions

Pre-actions specify a list of actions that have to be performed before reading or writing the register. Post-actions specify a list of actions that have to be performed after reading or writing the register. Pre-actions and post-actions are introduced by the `pre` and `post` keywords, optionally preceded by a read or write modifier and followed by a list of actions. An action is the assignment of an identifier to a specific value. The identifier designates a variable, the field of a structure or the element of an array. If no pre-actions or post-actions are provided, they default to `pre {}` or `post {}` where no actions have to be performed. If pre-actions or post-actions are preceded by a modifier, their definitions apply only to the read or write part of the register definition.

5.1 Register Extensions

This section describes some language extensions that introduce some short-hand notations for register definitions.

Parameterized registers

A parameterized register is introduced by appending a list of parameters to the identifier name of the register being defined. The right-hand side of the definition is identical to a standard register definition except that the parameters can appear in the expressions. Parameterized registers can be seen as functions from a set of parameters to a register definition.

```
register reg(i : int(2)) = read base @ 0, pre {index = i} : bit[8];
```

Parameterized register applications

The parameterized register application of a register definition is replaced by the definition of the parameterized register where parameters have been replaced by specific values. If a component is specified in the parameterized register, it can not be redefined. That is, a component can not be specified both in the parameterized register definition and in the *register-attribute* part of the register definition. If the parameterized register application is preceded by a read or write modifier, the modifier is applied to all components of the parameterized register definition. Components that have the read and write modifiers are ignored.

```
register x_low  = reg(0), mask '****....';
```

6 Type Expressions

```
type-expr         ::=  typexpr-def
                   |   typexpr-spec
typexpr-def       ::=  { map (, map)* } ([integer-literal])?
typexpr-spec      ::=  (signed)? int {integer-ranges}
                   |   (signed)? int (integer-literal)
                   |   bool
                   |   ident
                   |   typexpr-spec[integer-literal]
map               ::=  (private)? ident arrow bit-pattern
                   |   (private)? ident(type-params) arrow algtype-def
type-params       ::=  type-param (, type-param)*
type-param        ::=  ident :  typexpr-spec
algtype-def       ::=  bit-pattern
                   |   bit-pattern # algtype-def
arrow             ::=  <= | => | <=>
bit-pattern       ::=  bits-literal
                   |   integer-literal
type-definition   ::=  (private)? type ident = type-expr
```

A *map-type* definition consists of a list of *map-type equations* and is declared by using the {*map* (, *map*)*} construct. It specifies an enumeration of identifiers that form the abstract values of the type being defined. A *map-type* can be thought of as enumerated type.

Map-type equation

A *map-type equation* is introduced by the identifier name of an abstract value, followed by whether a left (`<=`) or a right (`=>`) arrow and then a bits-literal or an integer-literal. The identifier on the left-hand side is the name of a value (*abstract value*) of the type being defined. The value on the right-hand side denotes the corresponding *concrete value* that is read or written to the corresponding variable. If the identifier name on the left-hand side is preceded by the `private` keyword, then the defined value is declared to be private to the Devil specification and will not be exported into the generated interface.

```
{
  ROTATE_IN_AEOI_CLEAR    => '000',
  NON_SPECIFIC            => '001',
  private NOP             => '010',
  SPECIFIC_EOI            => '011',
  ROTATE_IN_AEOI_SET      => '100',
  ROTATE_NON_SPECIFIC_EOI => '101',
  SET_PRIORITY            => '110',
  ROTATE_SPECIFIC_EOI     => '111'
};
```

One of the private value of a *map-type* must be used at least once. Abstract values denote values of the types used in the generated interface. Concrete values are the corresponding bit-strings that are read or written in the register of a device. A *map-type equation* specifies both abstract and concrete values and defines if the abstract value can be read or written. The left arrow (<=) is used to specify that the concrete value can be read, and that its interpretation must be the abstract value defined on the left-hand side. The right arrow (=>) specifies that the concrete value can be written. The left-right arrow (<=>) can be used as a short hand notation when a concrete value, associated to the same abstract value, can be read or written. As an example, the following *map-type equations*

```
X <= '10',
X => '10'
```

are equivalent to

```
X <=> '10'
```

Bit Patterns

If different concrete values are associated to the same abstract value in read map-type equations, then one can use a *bit-pattern* as a short hand notation. The `*` (star sign) in a bits-literal denotes any bit. As an example, the following map-type equations

```
X <= '10',
X <= '11'
```

are equivalent to

```
X <= '1*'
```

Note that patterns are allowed only for read map-type equations.

Type abbreviations

The *type-definition* rule defines the type identifier name as an abbreviation for the type expression on the right-hand side of = (equal sign). If the `type` keyword is preceded by the `private` keyword, the type is not exported to the generated interface. A type declared as private must be used at least once in the Devil specification where it has been defined.

Algebraic types

An algebraic type equation denotes an abstract value which has a concrete representation that depends on values of other types. As an example, assuming that type t_1 is defined as follows :

```
type t1 = {
   X => '10',
   Y => '11',
   Z => '01'
};
```

one can define type t_2 by

```
type t2 = {
  A (x : t1) => '10' # x # '00',
  B          => '110000'
};
```

Any Devil type can be used as an algebraic type parameter. The # (sharp sign) denotes a concatenation of bits that forms the concrete value. Note that there is no cost penalty in using algebraic types for writing values. The cost for reading depends on the type representation complexity.

7 Variables

variable-definition	::=	`(private)`$^?$ `variable` *ident* `=` *vardef*
vardef	::=	*var-registers* `(, ` *var-attribute*`)`* `:` *type-expr*
var-registers	::=	*var-regs* `(, ` *var-regs*`)`$^?$
var-regs	::=	$(rw)^?$ *register-bits*
register-bits	::=	*ident*
	|	*ident*`[`*integer-ranges*`]`
	|	*ident*`(`*exprs*`)`
	|	*ident*`(`*exprs*`)[`*integer-ranges*`]`
	|	*register-bits* `#` *register-bits*
var-attribute	::=	`volatile`
	|	$(rw)^?$ `trigger`
	|	$(rw)^?$ `trigger for` *trigexpr-paren*
	|	$(rw)^?$ `trigger except` *trigexpr-paren*
	|	$(rw)^?$ *serial-spec*
trigexpr-paren	::=	*trigexpr*
	|	`(`*trigexpr* `(, ` *trigexpr*`)`*`)`
trigexpr	::=	*integer-range*
	|	*ident*
	|	*boolean-literal*

A variable object provides an abstract interface to concrete register values and specifies the semantics of the data stored in registers. A variable definition has two optional parts: the read part and the write part. If a variable definition does not have a read (write) part, the register is said to be write-only (read-only), otherwise it is referred to be read-write. The read part of a variable definition contains the specification of four components: register-bits, serialization, volatile and trigger. The write part specifies three components: register-bits, serialization and trigger. The type specification is shared for both the read and write parts.

Visibility

A variable object is introduced by the `variable` keyword, followed by the identifier name of the variable being defined, and optionally preceded by the `private` keyword. When a variable is declared as private, its definition is not exported in the generated interface and must be used at least once within the devil program where its definition occurs. As an example, private variables are often declared to model indexes and used in pre-actions of indexed-registers.

```
private variable index = index_reg[6..5] : int(2);
```

Register-bits

The *register-bits* component specifies which bit of which register must be concatenated in order to obtain the bit-string that forms a device variable. The register-bits component is recursively defined as follows: (1) a register identifier name denotes all its bits; (2) a register identifier name followed by an integer range listed between square brackets denotes all bits which have a position number that appear in the list; (3) the expression $rb_1 \# rb_2$ denotes the concatenation (from left to right) of the bits specified by rb_1 and rb_2. Note that the bit number 0 of register *reg* is the less significant bit. One or two register-bits must be provided in a variable definition. The read or write part of a variable is defined if and only if a read or write register-bits definition is provided. If only one register-bits is provided, without a modifier, its definition is used for both the read and write parts.

Volatile

The `volatile` attribute is only attached to the read part of a variable definition. When a variable is declared as volatile, each read operation may produce a different value. If the volatile attribute is not provided, it defaults to non volatile.

Trigger

The trigger attribute is introduced by the `trigger` keyword, optionally preceded by a read or write modifier. When a variable is declared as trigger, each access to this variable triggers an action or a set of actions inside the device. For example, writing twice the same value does not produce the same result that one write operation. As a consequence, two trigger variables that map bits of the same register have to be grouped in a structure. When the trigger attribute is used for the write part of a variable definition, some specific values can sometimes cancel the trigger effect. These values can be specified by using the attributes `trigger for` and `trigger except`. If no trigger attribute is provided, it defaults to non trigger. If the trigger attribute is preceded by the read or write modifier, its definition applies only to the read or write part of the variable definition.

Trigger for ...

When the `trigger` keyword is followed by the `for` keyword and a list of expressions, the variable is declared as trigger only for values that match the specified list of expressions.

Trigger except ...

When the `trigger` keyword is followed by the `except` keyword and then by a list of expressions, the variable is declared as trigger only for values that do not match the specified list of expressions. Since all Devil types are finite, the set of values that cancel the trigger effect of a variable is also finite.

7.1 Structures

structure-definition	::=	`structure` *ident* `=` *structdef*
structdef	::=	`{` (*variable-definition*`;`)$^+$ `}` (*structure-attributes*)$^?$
structure-attributes	::=	*structure-attribute* (`,` *structure-attribute*)$^?$
structure-attribute	::=	(*rw*)$^?$ *serial-spec*

A structure is a collection of variable definitions.

```
structure it_status = {
  variable it_status_rst = read isr[7], volatile : status;
  variable it_status_rdc = read isr[6], volatile : status;
  variable it_status_cnt = read isr[5], volatile : status;
  variable it_status_ovw = read isr[4], volatile : status;
  variable it_status_txe = read isr[3], volatile : status;
  variable it_status_rxe = read isr[2], volatile : status;
  variable it_status_ptx = read isr[1], volatile : status;
  variable it_status_prx = read isr[0], volatile : status;
};
```

When more than one register is used for defining variables contained in the structure, the `serialized as` expression has to be provided. When a structure is read or written, the serial construction specifies in which order read or write operations are executed. When bits of a register are mapped to variables defined in the same structure, only one read or write operation is executed for this register.

```
structure init = {
   variable sngl = icw1[1] : { SINGLE => '1', CASCADED => '0'};
   variable ic4  = icw1[0] : bool;
   ...
   variable microprocessor = icw4[0] : { X8086 => '1', MCS80_85 => '0'};
} serialized as {
   icw1; icw2;
   if (sngl == SINGLE) icw3;
   if (ic4 == true) icw4;
};
```

7.2 Serialization

serial-spec	::=	`serialized as` *serial-desc*
serial-desc	::=	*ident*
	|	`{` *serial-def* (`;` *serial-def*)* `}`
serial-def	::=	*ident*
	|	`if (`*expr*`)` *serial-desc* (`else` *serial-desc*)$^?$

The `serialized as` construct is used to specify in which order registers of a variable or a structure have to be accessed. The `serialized` keyword is optionally preceded by a read or write modifier, in which case its definition is restricted to the read or write part of the variable or structure being defined. If no modifier is specified, the same serial definition is used for both the read and write part. If no serial definition is provided, one is deduced from the variable or structure definition. Identifiers used in the rules *serial-desc* ::= *ident* and *serial-def* ::= *ident* must refer to register names.

Sequence

The serial description r_1 ; r_2 specifies that register r_1 must be accessed first, followed by an access to register r_2.

Conditional

The serial description `if (`*expr*`)` s_1 `else` s_2 specifies conditionnal definitions (depending on runtime values of given variables) of register sequences.

> **Example:** The 8259A interrupt controller possesses various execution modes that depend on the hardware configuration (processor type, cascaded/single controller). The initialization of the controller is performed by writing to configuration variables defined over four initialization registers. In fact, the initialization sequence varies with the actual values of configuration variables. Additionally, three of the configuration registers (e.g., `icw2, icw3, icw4`) are mapped on a single port and their addressing is implicitly done by previously written configuration values. The following example shows how such an addressing mode can be specified in Devil: configuration variables are grouped together within the `init` structure whose register write operations are ordered using tests on variable values.

```
register icw1 = write base@0, mask '***1****' : bit[8];
register icw2 = write base@1 : bit[8];
register icw3 = write base@1 : bit[8];
register icw4 = write base@1, mask '000*****' : bit[8];

structure init = {
   variable sngl = icw1[1] : { SINGLE => '1', CASCADED => '0'};
   variable ic4  = icw1[0] : bool;
   ...
   variable microprocessor = icw4[0] : { X8086 => '1', MCS80_85 => '0'};
```

```
} serialized as {
   icw1; icw2;
   if (sngl == SINGLE) icw3;
   if (ic4 == true) icw4;
};
```

7.3 Variable Extensions

When all bits of a register are mapped to a single variable and when this register can be expressed with a parameterized register, one can write:

```
variable foo = reg(1) : int(8);
```

instead of

```
register r1  = reg(1);
variable foo = r1 : int(8);
```

This two constructions are strictly equivalent. A similar case is when all bits (of a register) that can be mapped are mapped to a single variable. As illustrated previously, it is possible to insert a parameterized register application in a variable definition, but it is also possible to extract bits of this defined register with the `[...]` construct. The Devil code below illustrates this construction:

```
variable foo = reg(1)[4..0] : int(4);
```

A Logitech-Busmouse Example

This section shows who to use Devil for the development of a device driver. Section **??** presents the Devil specification of the logitech busmouse. Section **??** shows the C interface produced by the taz compiler from the Devil specification. The C interface was generated using Taz version 0.4 with the following command line :

```
#> taz -o busmouse_dil.h busmouse.dil
```

Section **??** gives one possible use of the generated interface in a Linux C driver.

A.1 Devil Specification

```
/************************************************************************/
/*                                                                      */
/*                         Logitech Bus Mouse                           */
/*                                                                      */
/*            Laurent Reveillere, projet Compose, INRIA Rennes          */
/*                                                                      */
/*   Copyright (C) 2000 INRIA. All rights reserved.                     */
/*   Institut National de Recherche en Informatique et en Automatique.  */
/*                                                                      */
/************************************************************************/


device logitech_busmouse (base : bit[8] port @ {0..3})
{

  // index register
  register index_reg = write base @ 2, mask '1..00000' : bit[8];

  private variable index = index_reg[6..5] : int(2);


  // interrupt register
  register interruption = write base @ 2, mask '000.0000' : bit[8];

  variable interrupt = interruption[4] :
  {
    ENABLE  => '0',
    DISABLE => '1'
  };


  // parametrized x/y registers
```

```
  register reg(i : int(2)) = read base @ 0, pre {index = i} : bit[8];


  register x_low  = reg(0), mask '****....';
  register x_high = reg(1), mask '****....';
  register y_low  = reg(2), mask '****....';
  register y_high = reg(3), mask '...*....';

  structure mouste_state = {
     variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
     variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
     variable buttons = y_high[7..5], volatile : int(3);
  };

  // configuration register (CR)
  register cr = write base @ 3, mask '1001000.' : bit[8];

  variable config = cr[0] :
  {
    CONFIGURATION => '1',
    DEFAULT_MODE  => '0'
  };


  // signature register (sr)
  register sr = base @ 1 : bit[8];

  variable sig = sr, volatile, write trigger : int(8);
}
```

A.2 Taz output

```
/*
  Copyright (C) 2000 IRISA/INRIA
  University of Rennes - France
  Automatically generated by Devil
*/
/*
   Check that DEVIL_BOOL_TYPED flag is used only when DEVIL_DEBUG flag is true
*/
#ifdef DEVIL_BOOL_TYPED
#ifndef DEVIL_DEBUG
```

```
#error "Devil error: DEVIL_BOOL_TYPED can be used only when DEVIL_DEBUG is true"
#endif /* DEVIL_DEBUG */
#endif /* DEVIL_BOOL_TYPED */


/*
   If several Devil files have to be included, chech that they all use the
   same type level
*/
#ifdef DEVIL_DEBUG
#ifdef DEVIL_BOOL_TYPED
#ifdef __DEVIL_TYPE_LEVEL_0__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifdef __DEVIL_TYPE_LEVEL_1__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifndef __DEVIL_TYPE_LEVEL_2__
#define __DEVIL_TYPE_LEVEL_2__
#endif
#else /* DEVIL_BOOL_TYPED */
#ifdef __DEVIL_TYPE_LEVEL_0__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifdef __DEVIL_TYPE_LEVEL_2__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifndef __DEVIL_TYPE_LEVEL_1__
#define __DEVIL_TYPE_LEVEL_1__
#endif
#endif /* DEVIL_BOOL_TYPED */
#else  /* DEVIL_DEBUG */
#ifdef __DEVIL_TYPE_LEVEL_1__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifdef __DEVIL_TYPE_LEVEL_2__
#error "Devil error: The same type level must be provide for multiples inclusions"
#endif
#ifndef __DEVIL_TYPE_LEVEL_0_
#define __DEVIL_TYPE_LEVEL_0__
#endif
#endif /* DEVIL_DEBUG */


/*
   Devil standard macros
```

```
 */
#ifndef __STD_DEVIL__
#define __dil_cat(x, y) x ## y
#define __dil_xcat(x, y) __dil_cat(x, y)
#define __STD_DEVIL__
#endif /* __STD_DEVIL__ */


/*
  If the interface is used with DEVIL_NO_REF flag, set the device name
  if it is not done before
 */
#ifdef DEVIL_NO_REF
#ifndef dev_name
#define dev_name logitech_busmouse
#endif /* dev_name */
#endif /* DEVIL_NO_REF */


/*
  Remove previous definitions and define misc functions
 */
#undef __dil_dev__
#undef __dil_dev_comma__
#undef __dil_dev_param__
#undef __dil_dev_param_comma__
#undef __dil_func
#undef __dil_access_cache
#ifndef DEVIL_NO_REF
#define __dil_dev__ dev
#define __dil_dev_comma__ dev,
#define __dil_dev_param__ dev_logitech_busmouse *dev
#define __dil_dev_param_comma__ dev_logitech_busmouse *dev,
#define __dil_func(name) name
#define __dil_access_cache(x) __dil_xcat((*dev)., x)
#else
#define __dil_dev__
#define __dil_dev_comma__
#define __dil_dev_param__
#define __dil_dev_param_comma__
#define __dil_func(name) __dil_xcat(dev_name, __dil_xcat(_, name))
#define __dil_access_cache(x) __dil_xcat(__dil_xcat(__dil_func(cache), .), x)
#endif /* DEVIL_NO_REF */


/*
```

```
  How to access the value of a bool type
 */
#undef __dil_bool_field
#ifdef DEVIL_BOOL_TYPED
#define __dil_bool_field(x) x.val
#else
#define __dil_bool_field(x) x
#endif /* DEVIL_BOOL_TYPED */


/*
  How to access the value of a given type, except bool
 */
#undef __dil_type_field
#ifdef DEVIL_DEBUG
#define __dil_type_field(x) x.val
#else
#define __dil_type_field(x) x
#endif /* DEVIL_DEBUG */


/*
  Functions used to compare elements of a enumerated type
  Note that no type checking can be done here by the C compiler
 */
#undef dil_eq
#define dil_eq(x, y) (__dil_type_field(x) == __dil_type_field(y))
#undef DIL_SWITCH
#define DIL_SWITCH(x) switch (__dil_type_field(x))
#undef DIL_CASE
#define DIL_CASE(x) case (__dil_type_field(x))


/*
  Define the assert function that must be used to raise fatal error
 */
#ifndef dil_assert
#define dil_assert(c) { if (!(c)) \
        printk(KERN_DEBUG "Assertion failed : line %d in %s.", __LINE__, __FILE__); }
#endif /* dil_assert */


/*
  Define the boolean type : bool
 */
#ifdef DEVIL_BOOL_TYPED
```

```
#ifndef __DIL_BOOL_TYPED_DEFINED__
#undef bool_t
#undef true
#undef false
struct bool_t_ { u32 val; };
typedef struct bool_t_ bool_t;
static const bool_t true = {1};
static const bool_t false = {0};
#define __DIL_BOOL_TYPED_DEFINED__
#endif /* __DIL_BOOL_TYPED_DEFINED__ */
#else
#ifndef __DIL_BOOL_DEFINED__
#undef bool_t
#undef true
#undef false
enum bool_t_ {true = 1, false = 0};
typedef enum bool_t_ bool_t;
#define __DIL_BOOL_DEFINED__
#endif /* __DIL_BOOL_DEFINED__ */
#endif /* DEVIL_BOOL_TYPED */

#ifdef DEVIL_DEBUG
struct config_t_ { u32 val; };
typedef struct config_t_ config_t;
static const config_t DEFAULT_MODE = {0x0u};
static const config_t CONFIGURATION = {0x1u};
#else
enum config_t_ {DEFAULT_MODE = 0x0u, CONFIGURATION = 0x1u};
typedef enum config_t_ config_t;
#endif /* DEVIL_DEBUG */
#ifdef DEVIL_DEBUG
struct interrupt_t_ { u32 val; };
typedef struct interrupt_t_ interrupt_t;
static const interrupt_t DISABLE = {0x1u};
static const interrupt_t ENABLE = {0x0u};
#else
enum interrupt_t_ {DISABLE = 0x1u, ENABLE = 0x0u};
typedef enum interrupt_t_ interrupt_t;
#endif /* DEVIL_DEBUG */
struct mouste_state_
{
  u8 cache_get_x_high;
  u8 cache_get_x_low;
  u8 cache_get_y_high;
  u8 cache_get_y_low;
};
```

```
struct dev_logitech_busmouse_
{
  u32 __dil_base__;
  struct mouste_state_ cache_mouste_state;
};
typedef struct dev_logitech_busmouse_ dev_logitech_busmouse;
#ifdef DEVIL_NO_REF
dev_logitech_busmouse __dil_func(cache);
#endif /* DEVIL_NO_REF */
static inline void
__dil_func(init_logitech_busmouse) (__dil_dev_param_comma__ u32 base)
{
  __dil_access_cache (__dil_base__) = base;
}

#include <linux/malloc.h>
#ifndef DEVIL_NO_REF
static inline dev_logitech_busmouse *
new_logitech_busmouse (int priority, u32 base)
{
  dev_logitech_busmouse *dev;

  dev = (dev_logitech_busmouse *)kmalloc (sizeof(*dev), priority);
  __dil_func(init_logitech_busmouse) (__dil_dev_comma__ base);
  return dev;
}
#endif /* DEVIL_NO_REF */
static inline void
__dil_func(reg_set_interruption) (__dil_dev_param_comma__ u8 v)
{
  outb ((u8)v, __dil_access_cache (__dil_base__) + 2);
}

static inline u8
__dil_func(reg_get_sr) (__dil_dev_param__)
{
  u8 tmp_0;

  tmp_0 = inb (__dil_access_cache (__dil_base__) + 1);
  return tmp_0;
}

static inline void
__dil_func(reg_set_sr) (__dil_dev_param_comma__ u8 v)
{
  outb ((u8)v, __dil_access_cache (__dil_base__) + 1);
```

```
}

static inline void
__dil_func(reg_set_cr) (__dil_dev_param_comma__ u8 v)
{
  outb ((u8)v, __dil_access_cache (__dil_base__) + 3);
}

static inline void
__dil_func(reg_set_index_reg) (__dil_dev_param_comma__ u8 v)
{
  outb ((u8)v, __dil_access_cache (__dil_base__) + 2);
}

static inline void
__dil_func(set_index) (__dil_dev_param_comma__ u8 v)
{
  u8 tmp_v;
  u8 tmp_0;

  tmp_v = v;
  tmp_0 = 0x80u | (tmp_v & 0x3u) << 5;
  __dil_func(reg_set_index_reg) (__dil_dev_comma__ tmp_0);
}

static inline u8
__dil_func(get_buttons) (__dil_dev_param__)
{
  u8 tmp_v;
  u8 v;
  u8 tmp_0;

  tmp_0 = (__dil_access_cache (cache_mouste_state)).cache_get_y_high;
  tmp_v = (tmp_0 & 0xe0u) >> 5;
  v = tmp_v;
  return v;
}

static inline void
__dil_func(set_config) (__dil_dev_param_comma__ config_t v)
{
  u8 tmp_v;
  u8 tmp_0;

  tmp_v = (u8)__dil_type_field (v);
  tmp_0 = 0x90u | tmp_v & 0x1u;
```

```
  __dil_func(reg_set_cr) (__dil_dev_comma__ tmp_0);
}

static inline s8
__dil_func(get_dx) (__dil_dev_param__)
{
  u8 tmp_v;
  s8 v;
  u8 tmp_0;
  u8 tmp_1;

  tmp_0 = (__dil_access_cache (cache_mouste_state)).cache_get_x_high;
  tmp_1 = (__dil_access_cache (cache_mouste_state)).cache_get_x_low;
  tmp_v = (tmp_0 & 0xfu) << 4 | tmp_1 & 0xfu;
  v = tmp_v;
  return v;
}

static inline s8
__dil_func(get_dy) (__dil_dev_param__)
{
  u8 tmp_v;
  s8 v;
  u8 tmp_0;
  u8 tmp_1;

  tmp_0 = (__dil_access_cache (cache_mouste_state)).cache_get_y_high;
  tmp_1 = (__dil_access_cache (cache_mouste_state)).cache_get_y_low;
  tmp_v = (tmp_0 & 0xfu) << 4 | tmp_1 & 0xfu;
  v = tmp_v;
  return v;
}

static inline void
__dil_func(set_interrupt) (__dil_dev_param_comma__ interrupt_t v)
{
  u8 tmp_v;
  u8 tmp_0;

  tmp_v = (u8)__dil_type_field (v);
  tmp_0 = (tmp_v & 0x1u) << 4;
  __dil_func(reg_set_interruption) (__dil_dev_comma__ tmp_0);
}

static inline u8
__dil_func(get_sig) (__dil_dev_param__)
```

```
{
  u8 tmp_v;
  u8 v;
  u8 tmp_0;

  tmp_0 = __dil_func(reg_get_sr) (__dil_dev__);
  tmp_v = tmp_0 & 0xffu;
  v = tmp_v;
  return v;
}

static inline void
__dil_func(set_sig) (__dil_dev_param_comma__ u8 v)
{
  u8 tmp_v;
  u8 tmp_0;

  tmp_v = v;
  tmp_0 = tmp_v & 0xffu;
  __dil_func(reg_set_sr) (__dil_dev_comma__ tmp_0);
}

static inline u8
__dil_func(reg_get_y_low) (__dil_dev_param__)
{
  u8 tmp_0;

  __dil_func(set_index) (__dil_dev_comma__ 2);
  tmp_0 = inb (__dil_access_cache (__dil_base__));
  return tmp_0;
}

static inline u8
__dil_func(reg_get_x_high) (__dil_dev_param__)
{
  u8 tmp_0;

  __dil_func(set_index) (__dil_dev_comma__ 1);
  tmp_0 = inb (__dil_access_cache (__dil_base__));
  return tmp_0;
}

static inline u8
__dil_func(reg_get_y_high) (__dil_dev_param__)
{
  u8 tmp_0;
```

```
  __dil_func(set_index) (__dil_dev_comma__ 3);
  tmp_0 = inb (__dil_access_cache (__dil_base__));
  return tmp_0;
}

static inline u8
__dil_func(reg_get_x_low) (__dil_dev_param__)
{
  u8 tmp_0;

  __dil_func(set_index) (__dil_dev_comma__ 0);
  tmp_0 = inb (__dil_access_cache (__dil_base__));
  return tmp_0;
}

static inline void
__dil_func(get_mouste_state) (__dil_dev_param__)
{
  (__dil_access_cache (cache_mouste_state)).cache_get_x_high = \
   __dil_func(reg_get_x_high) (__dil_dev__);
  (__dil_access_cache (cache_mouste_state)).cache_get_x_low  = \
   __dil_func(reg_get_x_low) (__dil_dev__);
  (__dil_access_cache (cache_mouste_state)).cache_get_y_high = \
   __dil_func(reg_get_y_high) (__dil_dev__);
  (__dil_access_cache (cache_mouste_state)).cache_get_y_low  = \
   __dil_func(reg_get_y_low) (__dil_dev__);
}
#undef DEVIL_NO_REF
#undef dev_name
```

A.3 C Driver

- Code to write at the begining of the file :

```
#define DEVIL_NO_REF /* no reference, only one mouse at a time */
#define dev_name bm  /* prefix to use */
#define DEVIL_DEBUG  /* use strong typing */
#include "busmouse_dil.h"
```

- Code to write at the begining of the init function :

```
bm_init_logitech_busmouse (LOGIBM_BASE);
```

- Example of code to write for accessing device variables :

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
outb(MSE_READ_Y_LOW, MSE_CONTROL_PORT );
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
buttons = ((buttons >> 5) & 0x07);
```

(a) Original C code

```
bm_get_mouse_state();

dx = bm_get_dx();
dy = bm_get_dy();
buttons = bm_get_buttons();
```

(b) Devil-based C code

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-0803