

# GenoM Manual

---

For GenoM version 3

Revision 2.99.14, updated October 2012

**Sara Fleury**

**Matthieu Herrb** [matthieu.herrb@laas.fr](mailto:matthieu.herrb@laas.fr)

**Anthony Mallet** [anthony.mallet@laas.fr](mailto:anthony.mallet@laas.fr)

**Cédric Pasteur**

---

GenoM3 is copyright © 2009-2012 LAAS/CNRS. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

genom-pcpp is copyright © 2004,2010 Anders Magnusson ([ragge@ludd.luth.se](mailto:ragge@ludd.luth.se)). All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Component model</b>	<b>3</b>
<b>3</b>	<b>GenoM overview</b>	<b>5</b>
<b>4</b>	<b>A minimal example</b>	<b>7</b>
<b>5</b>	<b>Input file format</b>	<b>9</b>
5.1	Preprocessing	9
5.1.1	#line directives	9
5.1.2	#pragma requires directives	10
5.1.3	#pragma provides directives	10
5.2	Elements of a GenoM3 specification	10
5.3	Component declaration	10
5.4	Interface declaration	12
5.5	IDS declaration	12
5.6	Task declaration	12
5.7	Port declaration	13
5.8	Attribute declaration	13
5.9	Service declaration	13
5.10	Service parameters	14
5.11	Codel declaration	14
5.12	Module declaration	15
5.13	Constant declaration	15
5.14	Type declaration	15
5.15	Type specification	15
5.16	Identifiers and reserved keywords	16
5.17	Grammar reference	17
<b>6</b>	<b>GenoM IDL mappings</b>	<b>23</b>
6.1	C mappings	23
6.1.1	Scoped names	23
6.1.2	Mapping for constants	23
6.1.3	Mapping for basic data types	23
6.1.4	Mapping for enumerated types	24
6.1.5	Mapping for strings	24
6.1.6	Mapping for arrays	24
6.1.7	Mapping for structure types	25
6.1.8	Mapping for union types	25
6.1.9	Mapping for sequence types	25
6.1.10	Mapping for port types	26
6.2	C++ mappings	27

6.2.1	Scoped names .....	27
6.2.2	Mapping for constants .....	27
6.2.3	Mapping for basic data types .....	27
6.2.4	Mapping for enumerated types .....	27
6.2.5	Mapping for strings .....	28
6.2.6	Mapping for arrays .....	28
6.2.7	Mapping for structure types .....	28
6.2.8	Mapping for union types .....	29
6.2.9	Mapping for sequence types .....	29
<b>7</b>	<b>Running genom .....</b>	<b>31</b>
7.1	Description .....	31
7.2	General options .....	31
7.3	Template options .....	33
7.4	Environment variables .....	33
<b>8</b>	<b>Templates .....</b>	<b>35</b>
8.1	The TCL backend .....	35
8.1.1	Runtime template information .....	35
	Source additional template code .....	35
	Generate template content .....	35
	Create symbolic links .....	36
	Define template options .....	36
	Template dependencies .....	36
	Retrieve options passed to templates .....	36
	Define template help string .....	37
	Print runtime information .....	37
	Abort template processing .....	37
8.1.2	Backend configuration .....	37
	Engine output configuration .....	37
	Automatic merge of generated content .....	38
	Change output directory .....	38
	Get current output directory .....	38
8.1.3	Input file information .....	38
	Genom program path and command line .....	38
	Template path and directories .....	39
	Input file name and path .....	39
	Process additional input .....	40
	Data type definitions from the specification .....	40
	Components definitions from the specification .....	40
8.1.4	Language dependent code generation .....	40
	Target programming language .....	40
	Canonical object name .....	41
	Generate comment strings .....	41
	Canonical file extension .....	41
	Generate indented text .....	41
	Generate filler string .....	41
	Chop blocks of text .....	42
	IDL type language mapping .....	42
	Code for type declarations .....	42
	Code for variable addresses .....	42
	Code for dereferencing variables .....	42
	Code for declaring functions arguments .....	43

Code for passing functions arguments.....	43
Code for accessing structure members .....	43
Code for declaring codel signatures .....	43
Code for calling codels .....	44
<b>Indices.....</b>	<b>45</b>
Index of concepts .....	45
Index of TCL backend procedures .....	46



# Introduction





# Component model



## Gen<sup>o</sup>M overview



## **A minimal example**



# Input file format

This chapter describes the **GenoM3** Input File Format (**dotgen**) semantics and gives the syntax for **dotgen** grammatical constructs. **dotgen** is the language used to formally describe a **GenoM** component in terms of services and data types it provides. A description written in **dotgen** completely defines the interface and the internals of a component.

A description of the **dotgen** preprocessing is presented in [Section 5.1 \[Preprocessing\], page 9](#). The grammar is presented in [Section 5.17 \[Grammar reference\], page 17](#). Associated semantics is described in the rest of this chapter either in place or through references to other sub sections of this chapter.

A source file containing a **dotgen** component specification must have a **‘.gen’** extension. The description of the **dotgen** grammar uses a syntax notation that is similar to EBNF (Extended Backus-Naur Format). The following table lists the symbols used in this format and their meaning.

Symbol	Meaning
<code>::=</code>	Definition.
<code> </code>	Alternation.
<code>text</code>	Nonterminals.
<code>"text"</code>	Terminals.
<code>( ... )</code>	Grouping.
<code>{ ... }</code>	Repetition: may occur zero or any number of times.
<code>[ ... ]</code>	Option: may occur zero or one time.

Table 5.1: **dotgen** EBNF symbols

## 5.1 Preprocessing

A **dotgen** specification consists of one or more files that are preprocessed. The preprocessing is controlled by directives introduced by lines having **#** as the first character other than white space. Preprocessor directives have their own syntax (namely, the C preprocessor syntax), independent of the **dotgen** language and not entirely described in this document. For instance, see *The C preprocessor*<sup>1</sup> for a comprehensive documentation.

The C preprocessor used by **GenoM** is invoked as a separate process from **libexec/genom-pcpp** by default. This is the C preprocessor from the **pcc** project (<http://pcc.ludd.ltu.se/>). This can be changed by setting the environment variable **CPP**, See [Section 7.4 \[Environment variables\], page 33](#). However, note that if you change the default, you will lose some functionalities of the **genom-pcpp** program. See [Section 5.1.2 \[#pragma requires\], page 10](#).

The primary use of the preprocessing facility is to include definitions (especially type definitions) from other **dotgen** specifications. Directives may appear anywhere in the source file but are not seen nor interpreted by **GenoM**. For instance, text in files included with a **#include** directive is treated as if it appeared in the including file. However, the preprocessor outputs some information, also as **#** directives, that are available to **GenoM**. These are described in the following sections.

### 5.1.1 #line directives

Line directives are inserted by the C preprocessor. They inform the **GenoM** parser about the line number and file name where each token was defined. Line directives can also be inserted manually to achieve special effect on error reporting or similar.

<sup>1</sup> <http://gcc.gnu.org/onlinedocs/cpp/>

A line directive starts with the `#line` token, followed by the current line number, the file name of the source file and an optional numeric flag. The numeric flag is never used by `GenoM`. Its meaning depends on the C preprocessor used.

```
#line linenum [ "filename" [flag] ]
```

The file name and line number replace the current location information. Line number is automatically incremented after each `\n`.

### 5.1.2 #pragma requires directives

`#pragma requires` is recognized by *both* the `genom-pcpp` preprocessor and `GenoM`. This can be used to indicate an external dependency on a package using the `pkg-config` utility (see <http://www.freedesktop.org/wiki/Software/pkg-config>).

```
#pragma requires "package [ >= version ]"
```

`#pragma requires` accepts a string argument in the form `package [ >= version ]`. `genom-pcpp` interprets it by running `pkg-config --cflags` on the string argument. It then adds the resulting `-I` and `-D` flags as if they had been passed on the command line (see [Section 7.2 \[General options\]](#), page 31). Note that the flags are added *at the current processing location*, so they do not influence already preprocessed input. The `pkg-config` utility is found in `PATH`, or via the `PKG_CONFIG` environment variable if defined (see [Section 7.4 \[Environment variables\]](#), page 33).

`GenoM` also recognizes `#pragma requires`. It handles it by adding the string argument to the `require` property of *all* the components defined in a specification (see [Section 5.3 \[Component declaration\]](#), page 10).

### 5.1.3 #pragma provides directives

## 5.2 Elements of a GenoM3 specification

A dotgen specification consists of one or more statements. Statements are either `GenoM` statements, IDL statements. `cpp` directives (see [Section 5.1 \[Preprocessing\]](#), page 9) are handled at the lexical level and do not interfere with the specification grammar.

```
(1) specification      ::= { statement }
(2) statement          ::= component
                        | interface
                        | idl-statement

(4) idl-statement      ::= module
                        | const-dcl ";"
                        | type-dcl ";"
```

Definitions are named by the mean of identifiers, see [Section 5.16 \[Reserved keywords\]](#), page 16.

A `GenoM` statement defines components (see [Section 5.3 \[Component declaration\]](#), page 10) or interfaces (see [Section 5.4 \[Interface declaration\]](#), page 12).

An IDL statement defines types (see [Section 5.14 \[Type declaration\]](#), page 15), constants (see [Section 5.13 \[Constant declaration\]](#), page 15) or IDL modules containing types and constants (see [Section 5.12 \[Module declaration\]](#), page 15). The syntax follows closely the subset the OMG IDL specification corresponding to type and constants definitions (see Chapter 7 of *CORBA specification, Object Management Group, version 3.1. Part I: CORBA interfaces*). Note that this subset of the dotgen grammar is not in any manner tied to OMG IDL and may diverge from future OMG specifications.

## 5.3 Component declaration

```
(5) component          ::= "component" component-name component-body ";"
```



```

(6) component-name      ::= identifier
(13) component-body     ::= [ "{" exports "}" ]

(14) exports            ::= { export }
(15) export             ::= idl-statement
                           | property
                           | ids
                           | task
                           | port
                           | attribute
                           | service

(7) component-property  ::= ( "doc" string-literals | "version"
                             string-literals | "lang" string-literals |
                             "email" string-literals | "requires"
                             string-list | "codels-require" string-list
                             | "clock-rate" const-expr time-unit |
                             "provides" interface-list | "uses"
                             interface-list ) ";"

```

A component declaration describes an instance of the **GenoM** component model. It is defined by a unique name (an identifier) that also defines an IDL scope for any embedded types.

Components export objects from the **GenoM** component model, namely: **IDS** (see [Section 5.5 \[IDS declaration\]](#), page 12), **tasks** (see [Section 5.6 \[Task declaration\]](#), page 12), **ports** (see [Section 5.7 \[Port declaration\]](#), page 13), **attributes** (see [Section 5.8 \[Attribute declaration\]](#), page 13) or **services** (see [Section 5.9 \[Service declaration\]](#), page 13).

Components may also define new types *via* IDL statements. These types will be defined within the component scope.

A number of properties can be attached to a component:

<b>doc</b>	A string that describes the functionality of the component.
<b>version</b>	The component version number, as a string
<b>lang</b>	The programming language of the codels interface.
<b>email</b>	A string containing the e-mail address of the author of the component.
<b>requires</b>	A list of dependencies of the component (see <a href="#">Section 5.1.2 [#pragma requires]</a> , page 10). Each string should contain a package name in <b>pkg-config</b> format.
<b>codels-requires</b>	A list of dependencies of the codels. Each string should contain a package name in <b>pkg-config</b> format.
<b>clock-rate</b>	The period of the internal component clock. It is usually not necessary to define it explicitly. If the component defines periodic task, the component clock period will be automatically computed as the greatest common divisor of the period of all periodic tasks.
<b>provides</b>	A list of interfaces (see <a href="#">Section 5.4 [Interface declaration]</a> , page 12) that the component implements. All objects from the interface are imported as-is into the component description. Ports and services may be further refined once imported, typically by defining codels (see <a href="#">Section 5.11 [Codel declaration]</a> , page 14) that implement the services.
<b>uses</b>	A list of interfaces (see <a href="#">Section 5.4 [Interface declaration]</a> , page 12) that the component uses. Ports are imported in the opposite direction (e.g. a <b>port out</b> is imported as a <b>port</b>

in. Services are imported as **remote** objects that can be accessed *via* codel parameters (see [Section 5.11 \[Codel declaration\]](#), page 14). Other objects are imported as-is.

## 5.4 Interface declaration

```
(8) interface          ::= "interface" interface-scope component-body ";"
(9) interface-scope    ::= identifier

(11) interface-property ::= "extends" interface-list ";"
```

An interface declaration is basically the same as a component declaration (see [Section 5.3 \[Component declaration\]](#), page 10) but is meant to be shared between several components. Although any object can be defined in an interface, it will typically only declare service prototypes and ports that are to be **provided** or **used** by components.

In addition to regular component properties, an interface can also define the following properties:

**extends**      A list of interfaces that are imported as-is into the current one. All objects from the extended interfaces appear as if they had been defined in the extending interface.

## 5.5 IDS declaration

```
(16) ids                ::= ids-name "{" member-list "}" ";"
(17) ids-name           ::= "ids"
```

## 5.6 Task declaration

```
(18) task                ::= "task" identifier opt-properties ";"
(26) opt-properties      ::= [ "{" properties "}" ]
(27) properties          ::= { property }

(19) task-property       ::= ( "period" const-expr time-unit | "delay"
                               const-expr time-unit | "priority"
                               positive-int-const | "scheduling" "real-time" |
                               "stack" positive-int-const size-unit ) ";"
(30) codel-property      ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
```

Tasks define an execution context suitable for running *activities* (see [Section 5.3 \[Component declaration\]](#), page 10). A task may define a state machine and associated codels (see [Section 5.11 \[Codel declaration\]](#), page 14). The state machine starts in the **start** state when the task is created during component initialization.

Tasks are named can also define the following properties:

**period**      The granularity of the codel scheduler. Periodic task will sequence the codels they manage at that frequency.

**delay**      The delay from the beginning of each period after which codels are run. This can be used to delay two tasks running at the same period in the same component.

**priority**    Can be used to prioritize different tasks within the same component.

**scheduling real-time**  
This indicates that the task requires real-time scheduling. This may not be supported by all templates.

**stack**      Defines the required stack size for this task. The stack size should be big enough to run all codels that the task manages.

## 5.7 Port declaration

```

(20) port                ::= "port" opt-multiple port-dir type-spec
                             identifier ";"
(22) opt-multiple         ::= [ "multiple" ]
(21) port-dir            ::= "in"
                             | "out"

```

Ports implement the data flow between components as a publish/subscribe model. Ports have a name and a type and can be either out (for publishing data) or in (for subscribing to a sibling out port).

The optional `multiple` qualifier defines a dynamic list of ports of the given type, indexed by strings. In this case, ports are created or destroyed dynamically by the codels.

## 5.8 Attribute declaration

```

(23) attribute           ::= "attribute" identifier "(" attribute-parameters
                             ")" opt-properties ";"
(32) attribute-parameters ::= [ { attribute-parameter "," }
                             attribute-parameter ]
(33) attribute-parameter ::= parameter-dir parameter-variable
                             opt-initializer
(26) opt-properties      ::= [ "{" properties "}" ]
(27) properties          ::= { property }
(29) service-property    ::= ( "task" identifier | "interrupts"
                             identifier-list | "before" identifier-list |
                             "after" identifier-list | "validate" codel )
                             ";"

```

## 5.9 Service declaration

```

(24) service             ::= service-kind identifier "(" service-parameters
                             ")" opt-properties ";"
(25) service-kind        ::= "function"
                             | "activity"
(34) service-parameters ::= [ { service-parameter "," } service-parameter ]
(35) service-parameter   ::= parameter-dir type-spec declarator
                             opt-initializer
(26) opt-properties      ::= [ "{" properties "}" ]
(27) properties          ::= { property }
(28) property            ::= component-property
                             | interface-property
                             | task-property
                             | service-property
                             | codel-property
                             | throw-property
(29) service-property    ::= ( "task" identifier | "interrupts"
                             identifier-list | "before" identifier-list |
                             "after" identifier-list | "validate" codel )
                             ";"

```

```

(30) codel-property      ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
(43) opt-async           ::= [ "async" ]

```

## 5.10 Service parameters

```

(33) attribute-parameter ::= parameter-dir parameter-variable
                             opt-initializer
(35) service-parameter  ::= parameter-dir type-spec declarator
                             opt-initializer

(39) parameter-dir       ::= "in"
                             | "out"
                             | "inout"
(40) parameter-variable  ::= identifier
                             | parameter-variable "." identifier
                             | parameter-variable "[" positive-int-const "]"
(45) opt-initializer     ::= [ "=" initializer ]
(46) initializers        ::= [ { initializer "," } initializer ]
(47) initializer         ::= initializer-value
                             | ":" string-literals
                             | initializer-value ":" string-literals
(48) initializer-value   ::= const-expr
                             | "{" initializers "}"
                             | "[" positive-int-const "]" "=" const-expr
                             | "[" positive-int-const "]" "=" "{" initializers
                               "}"
                             | "." identifier "=" const-expr
                             | "." identifier "=" "{" initializers "}"

```

## 5.11 Codel declaration

```

(41) codel               ::= identifier "(" codel-parameters ")"
(42) fsm-codel           ::= event-list identifier "(" codel-parameters ")"
                             "yields" identifier-list
(36) codel-parameters    ::= [ { codel-parameter "," } codel-parameter ]
(37) codel-parameter     ::= opt-parameter-src parameter-dir (
                             parameter-variable
                             | parameter-variable ":" identifier | ":"
                               identifier )
(43) opt-async           ::= [ "async" ]
(38) opt-parameter-src   ::= [ "ids" | "local" | "port" | "remote" ]
(39) parameter-dir       ::= "in"
                             | "out"
                             | "inout"
(40) parameter-variable  ::= identifier
                             | parameter-variable "." identifier
                             | parameter-variable "[" positive-int-const "]"
(44) event-list          ::= "<" identifier-list ">"

```

## 5.12 Module declaration

A module definition satisfies the following syntax:

```
(49) module                ::= "module" module-name "{" module-body "}" ";"
(50) module-name           ::= identifier
(51) module-body           ::= [ idl-statements ]
(3) idl-statements        ::= { idl-statement } idl-statement
```

The only effect of a module is to scope IDL identifiers. It is similar to a C++ or Java namespace; it is considered good practice to enclose your type definitions inside a module definition to prevent name clashes between components.

## 5.13 Constant declaration

```
(52) const-dcl             ::= "const" const-type identifier "=" const-expr
(53) const-type            ::= integer-type
                               | char-type
                               | boolean-type
                               | floating-pt-type
                               | octet-type
                               | string-type
                               | named-type
```

## 5.14 Type declaration

Type declarations define new data types and associate a name (an identifier) with it. The `typedef` keyword can be used to name an existing type. The constructed types `struct`, `union` and `enum` also name the type they define. The syntax is the following:

```
(54) type-dcl              ::= constructed-type
                               | "typedef" alias-list
                               | forward-dcl
(55) constructed-type      ::= struct-type
                               | union-type
                               | enum-type
(56) alias-list            ::= ( type-spec | alias-list "," ) declarator
```

## 5.15 Type specification

A type specification is the description of a type. It can be used in a `typedef` construct or anywhere a typed value is expected.

```
(65) type-spec             ::= simple-type-spec
                               | constructed-type-spec
(66) simple-type-spec      ::= base-type-spec
                               | template-type-spec
                               | named-type
(69) base-type-spec        ::= boolean-type
                               | integer-type
                               | floating-pt-type
                               | char-type
                               | octet-type
                               | any-type
```

```

(70) template-type-spec ::= sequence-type
                           | string-type
                           | fixed-type

(67) constructed-type-spec ::= constructed-type

(68) named-type ::= scoped-name

(101) scoped-name ::= [ [ scoped-name ] "::" ] identifier

(61) declarator ::= simple-declarator
                  | array-declarator

(62) simple-declarator ::= identifier

```

## 5.16 Identifiers and reserved keywords

An identifier is a sequence of ASCII alphabetic, digit, and underscore (`_`) characters. The first character must be an ASCII alphabetic character.

```

(117) identifier ::= "[A-Za-z-][A-Za-z0-9-]*"
                  | "s"
                  | "ms"
                  | "us"
                  | "k"
                  | "m"
                  | "real-time"
                  | "interface"
                  | "component"
                  | "ids"
                  | "attribute"
                  | "function"
                  | "activity"
                  | "version"
                  | "lang"
                  | "email"
                  | "requires"
                  | "codelets-require"
                  | "clock-rate"
                  | "task"
                  | "task"
                  | "period"
                  | "delay"
                  | "priority"
                  | "scheduling"
                  | "stack"
                  | "codelet"
                  | "validate"
                  | "yields"
                  | "throws"
                  | "doc"
                  | "interrupts"
                  | "before"
                  | "after"
                  | "handle"
                  | "port"
                  | "in"

```

```

| "out"
| "inout"
| "local"
| "async"
| "remote"
| "extends"
| "provides"
| "uses"
| "multiple"
(118) identifier-list ::= { identifier "," } identifier

```

Words that are reserved keywords in the dotgen language are valid identifiers where their use is not ambiguous.

## 5.17 Grammar reference

```

(1) specification ::= { statement }
(2) statement ::= component
| interface
| idl-statement
(3) idl-statements ::= { idl-statement } idl-statement
(4) idl-statement ::= module
| const-dcl ";"
| type-dcl ";"
(5) component ::= "component" component-name component-body ";"
(6) component-name ::= identifier
(7) component-property ::= ( "doc" string-literals | "version"
string-literals | "lang" string-literals |
"email" string-literals | "requires"
string-list | "codels-require" string-list
| "clock-rate" const-expr time-unit |
"provides" interface-list | "uses"
interface-list ) ";"
(8) interface ::= "interface" interface-scope component-body ";"
(9) interface-scope ::= identifier
(10) interface-name ::= identifier
(11) interface-property ::= "extends" interface-list ";"
(12) interface-list ::= { interface-name "," } interface-name
(13) component-body ::= [ "{" exports "}" ]
(14) exports ::= { export }
(15) export ::= idl-statement
| property
| ids
| task
| port
| attribute
| service
(16) ids ::= ids-name "{" member-list "}" ";"
(17) ids-name ::= "ids"

```

```

(18) task                ::= "task" identifier opt-properties ";"
(19) task-property       ::= ( "period" const-expr time-unit | "delay"
                                const-expr time-unit | "priority"
                                positive-int-const | "scheduling" "real-time" |
                                "stack" positive-int-const size-unit ) ";"
(20) port                ::= "port" opt-multiple port-dir type-spec
                                identifier ";"
(21) port-dir            ::= "in"
                                | "out"
(22) opt-multiple         ::= [ "multiple" ]
(23) attribute           ::= "attribute" identifier "(" attribute-parameters
                                ")" opt-properties ";"
(24) service             ::= service-kind identifier "(" service-parameters
                                ")" opt-properties ";"
(25) service-kind        ::= "function"
                                | "activity"
(26) opt-properties      ::= [ "{" properties "}" ]
(27) properties          ::= { property }
(28) property            ::= component-property
                                | interface-property
                                | task-property
                                | service-property
                                | codel-property
                                | throw-property
(29) service-property     ::= ( "task" identifier | "interrupts"
                                identifier-list | "before" identifier-list |
                                "after" identifier-list | "validate" codel )
                                ";"
(30) codel-property      ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
(31) throw-property      ::= "throws" identifier-list ";"
(32) attribute-parameters ::= [ { attribute-parameter "," }
                                attribute-parameter ]
(33) attribute-parameter ::= parameter-dir parameter-variable
                                opt-initializer
(34) service-parameters  ::= [ { service-parameter "," } service-parameter ]
(35) service-parameter   ::= parameter-dir type-spec declarator
                                opt-initializer
(36) codel-parameters    ::= [ { codel-parameter "," } codel-parameter ]
(37) codel-parameter     ::= opt-parameter-src parameter-dir (
                                parameter-variable
                                | parameter-variable "::" identifier | "::"
                                identifier )
(38) opt-parameter-src   ::= [ "ids" | "local" | "port" | "remote" ]
(39) parameter-dir       ::= "in"
                                | "out"
                                | "inout"

```



```

(40) parameter-variable ::= identifier
                           | parameter-variable "." identifier
                           | parameter-variable "[" positive-int-const "]"

(41) code1                ::= identifier "(" code1-parameters ")"

(42) fsm-code1            ::= event-list identifier "(" code1-parameters ")"
                           "yields" identifier-list

(43) opt-async            ::= [ "async" ]

(44) event-list           ::= "<" identifier-list ">"

(45) opt-initializer      ::= [ "=" initializer ]

(46) initializers         ::= [ { initializer "," } initializer ]

(47) initializer          ::= initializer-value
                           | ":" string-literals
                           | initializer-value ":" string-literals

(48) initializer-value    ::= const-expr
                           | "{" initializers "}"
                           | "[" positive-int-const "]" "=" const-expr
                           | "[" positive-int-const "]" "=" "{" initializers
                           "}"
                           | "." identifier "=" const-expr
                           | "." identifier "=" "{" initializers "}"

(49) module               ::= "module" module-name "{" module-body "}" ";";

(50) module-name          ::= identifier

(51) module-body          ::= [ idl-statements ]

(52) const-dcl            ::= "const" const-type identifier "=" const-expr

(53) const-type           ::= integer-type
                           | char-type
                           | boolean-type
                           | floating-pt-type
                           | octet-type
                           | string-type
                           | named-type

(54) type-dcl             ::= constructed-type
                           | "typedef" alias-list
                           | forward-dcl

(55) constructed-type     ::= struct-type
                           | union-type
                           | enum-type

(56) alias-list           ::= ( type-spec | alias-list "," ) declarator

(57) struct-type          ::= "struct" scope-push-struct "{" member-list "}"

(58) union-type           ::= "union" scope-push-union "switch" "("
                           switch-type-spec ")" "{" switch-body "}"

(59) enum-type            ::= "enum" identifier "{" enumerator-list "}"

(60) forward-dcl          ::= ( "struct" | "union" ) identifier

(61) declarator           ::= simple-declarator
                           | array-declarator

(62) simple-declarator    ::= identifier

```



(90) switch-type-spec	::= integer-type   char-type   boolean-type   enum-type   named-type
(91) switch-body	::= { case } case
(92) member-list	::= { member ";" } member ";"
(93) member	::= ( type-spec   member "," ) declarator
(94) case	::= case-label-list type-spec declarator ";"
(95) case-label-list	::= { case-label } case-label
(96) case-label	::= ( "case" const-expr   "default" ) ":"
(97) enumerator-list	::= { enumerator "," } enumerator
(98) enumerator	::= identifier
(99) scope-push-struct	::= identifier
(100) scope-push-union	::= identifier
(101) scoped-name	::= [ [ scoped-name ] ":" ] identifier
(102) const-expr	::= or-expr
(103) positive-int-const	::= const-expr
(104) or-expr	::= { xor-expr " " } xor-expr
(105) xor-expr	::= { and-expr "^" } and-expr
(106) and-expr	::= { shift-expr "&" } shift-expr
(107) shift-expr	::= { add-expr ( ">>"   "<<" ) } add-expr
(108) add-expr	::= { mult-expr ( "+"   "-" ) } mult-expr
(109) mult-expr	::= { unary-expr ( "*"   "/"   "%" ) } unary-expr
(110) unary-expr	::= [ "-"   "+"   "~" ] primary-expr
(111) primary-expr	::= literal   "(" const-expr ")"   named-type
(112) literal	::= "TRUE"   "FALSE"   integer-literal   "<float-literal>"   "<fixed-literal>"   "<char-literal>"   string-literals
(113) string-literals	::= { string-literal } string-literal
(114) string-list	::= { string-literals "," } string-literals
(115) time-unit	::= [ "s"   "ms"   "us" ]
(116) size-unit	::= [ "k"   "m" ]
(117) identifier	::= "[A-Za-z-][A-Za-z0-9-]*"   "s"   "ms"   "us"   "k"   "m"   "real-time"

```

| "interface"
| "component"
| "ids"
| "attribute"
| "function"
| "activity"
| "version"
| "lang"
| "email"
| "requires"
| "codels-require"
| "clock-rate"
| "task"
| "task"
| "period"
| "delay"
| "priority"
| "scheduling"
| "stack"
| "codel"
| "validate"
| "yields"
| "throws"
| "doc"
| "interrupts"
| "before"
| "after"
| "handle"
| "port"
| "in"
| "out"
| "inout"
| "local"
| "async"
| "remote"
| "extends"
| "provides"
| "uses"
| "multiple"

(118) identifier-list ::= { identifier "," } identifier

```

# GenoM IDL mappings

GenoM IDL is independent of the programming language used to implement the services and internals of a component. In order to use the GenoM generated source code, it is necessary for programmers to know how to access the service parameters and ports from their programming languages. This chapter defines the mapping of GenoM IDL constructs to the supported programming languages.

The mapping between GenoM IDL and a programming language diverges from the OMG CORBA standard. This is unfortunate, because this might lead to some confusion for the developers used to OMG CORBA, but it was necessary to define mappings well targetting real-time platforms. The design strategy that guided the definition of those mappings was to try to have contiguous memory segments, that do not require memory management primitives, for most of the data types. Only unbounded string and sequences do not follow this scheme.

GenoM currently implements mappings for the C and C++ languages. For the C language, See [Section 6.1 \[C mappings\], page 23](#). For the C++ language, see [Section 6.2 \[C++ mappings\], page 27](#).

## 6.1 C mappings

### 6.1.1 Scoped names

The C mappings always use the global name for a type or a constant. The C global name corresponding to a GenoM IDL global name is derived by converting occurrences of "::" to "\_" (an underscore) and eliminating the leading underscore.

### 6.1.2 Mapping for constants

In C, constants defined in dotgen are mapped to a C constant. For instance, the following IDL:

```
const long longint = 1;
const string str = "string example";
```

would map into

```
const uint32_t longint = 1;
const char *str = "string example";
```

The identifier can be referenced at any point in the user's code where a literal of that type is legal.

### 6.1.3 Mapping for basic data types

The basic data types have the mappings shown in the table below. Integer types use the C99 fixed size integer types as provided by the `stdint.h` standard header. Users do not have to include this header: the template mapping generation procedure output the appropriate `#include` directive along with the mappings for the integer types.

IDL	C
boolean	bool
unsigned short	uint16_t
short	int16_t
unsigned long	uint32_t
long	int32_t
unsigned long long	uint64_t
long long	int64_t
float	float
double	double
char	char
octet	uint8_t
any	type any not implemented yet

Table: Basic data types mappings in C

### 6.1.4 Mapping for enumerated types

The C mapping of an IDL `enum` type is an unsigned, 32 bits wide integer. Each enumerator in an `enum` is defined in an anonymous `enum` with an appropriate unsigned integer value conforming to the ordering constraints.

For instance, the following IDL:

```
module m {
  enum e {
    value1,
    value2
  };
};
```

would map, according to the scoped names rules, into

```
typedef uint32_t m_e;
enum {
  m_value1 = 0
  m_value2 = 1
};
```

### 6.1.5 Mapping for strings

Gen<sub>o</sub>M IDL bounded strings are mapped to nul terminated character arrays (i.e., C strings). Unbounded strings are mapped to a pointer on such a character array.

For instance, the following OMG IDL declarations:

```
typedef string unbounded;
typedef string<16> bounded;
```

would map into

```
typedef char *unbounded;
typedef char bounded[16];
```

### 6.1.6 Mapping for arrays

Gen<sub>o</sub>M IDL arrays map directly to C arrays. All array indices run from 0 to `size-1`.

For instance, the following IDL:

```
typedef long array[4][16];
```

would map into

```
typedef int32_t array[4][16];
```

### 6.1.7 Mapping for structure types

GenoM IDL structures map directly onto C `structs`. Note that these structures may potentially include padding.

For instance, the following IDL:

```
struct s {
    long a;
    long b;
};
```

would map into

```
typedef struct {
    int32_t a;
    int32_t b;
} s;
```

### 6.1.8 Mapping for union types

GenoM IDL unions map onto C `structs`. The discriminator in the enum is referred to as `_d`, the union itself is referred to as `_u`.

For instance, the following IDL:

```
union u switch(long) {
    case 1: long a;
    case 2: float b;
    default: char c;
};
```

would map into

```
typedef struct {
    int32_t _d;
    union {
        int32_t a;
        float b;
        char c;
    } _u;
} u;
```

### 6.1.9 Mapping for sequence types

GenoM IDL sequences mapping differ slightly for bounded or unbounded variations of the sequence. Both types maps onto a C `struct`, with a `_maximum`, `_length` and `_buffer` members.

For unbounded sequences, `buffer` points to a buffer of at most `_maximum` elements and containing `_length` valid elements. An additional member `_release` is a function pointer that can be used to release the storage associated to the `_buffer` and reallocate it. It is the responsibility of the user to maintain the consistency between those members.

For bounded sequences, `buffer` is an array of at most `_maximum` elements and containing `_length` valid elements. Since `_buffer` is an array, no memory management is necessary for this data type.

For instance, the following IDL:

```
typedef sequence<long> unbounded;
typedef sequence<long,16> bounded;
```

would map into

```
typedef struct {
    uint32_t _maximum, _length;
```

```

    int32_t *_buffer;
    void (*release)(void *_buffer);
} unbounded;

typedef struct {
    const uint32_t _maximum;
    uint32_t _length;
    int32_t _buffer[16];
} bounded;

```

### 6.1.10 Mapping for port types

Simple ports map onto an object-like C struct with a `data()`, `read()` or `write()` and `strerror()` function members. The `data()` function takes no parameter and returns a pointer on the current port data. Input ports may refresh their data by invoking the `read()` method, while output ports may publish new data by invoking the `write()` method. Both `read()` and `write()` return 0 on success, or an unsigned 32bits integer representing an error code. The `strerror()` method can be used to transform the error code into a user readable error message.

Ports defined with the `multiple` flag map onto a similar struct, with the difference that `data()`, `read()` and `write()` methods take an additional string (`const char *`) parameter representing the port element name. Multiple output ports have two additional `open()` and `close()` members (also accepting a single string parameter) that dynamically create or destroy ports.

For instance, the following IDL:

```

port in double in_port;
port multiple in double multi_in_port;
port out double out_port;
port multiple out double multi_out_port;

```

would map into

```

typedef struct {
    double * (*data)();
    uint32_t (*read)(void);
    const char * (*strerror)(uint32_t);
} in_port;

```

```

typedef struct {
    double * (*data)(const char *id);
    uint32_t (*read)(const char *id);
    const char * (*strerror)(uint32_t);
} multi_in_port;

```

```

typedef struct {
    double * (*data)();
    uint32_t (*write)(void);
    const char * (*strerror)(uint32_t);
} out_port;

```

```

typedef struct {
    double * (*data)(const char *id);
    uint32_t (*write)(const char *id);
    uint32_t (*open)(const char *id);
    uint32_t (*close)(const char *id);
    const char * (*strerror)(uint32_t);
} multi_out_port;

```



```
} multi_out_port;
```

## 6.2 C++ mappings

### 6.2.1 Scoped names

The C++ mappings for scoped names use C++ scopes. IDL modules are mapped to `namespaces`. For instance, the following IDL:

```
module m {
    const string str = "scoped string";
};
```

would map into

```
namespace m {
    const std::string str = "scoped string";
}
```

### 6.2.2 Mapping for constants

GenoM IDL constants are mapped to a C++ constant. For instance, the following IDL:

```
const long longint = 1;
const string str = "string example";
would map into
const int32_t longint = 1;
const std::string str = "string example";
```

### 6.2.3 Mapping for basic data types

The basic data types have the mappings shown in the table below. Integer types use the C99 fixed size integer types as provided by the `stdint.h` standard header (since the C++ `cstdint` header is not part of the C++ at the time of writing this document). Users do not have to include this header: the template mapping generation procedure output the appropriate `#include` directive along with the mappings for the integer types.

IDL	C++
boolean	bool
unsigned short	uint16_t
short	int16_t
unsigned long	uint32_t
long	int32_t
unsigned long long	uint64_t
long long	int64_t
float	float
double	double
char	char
octet	uint8_t
any	type any not implemented yet

Table: Basic data types mappings in C++

### 6.2.4 Mapping for enumerated types

The C++ mapping of an IDL `enum` type is the corresponding C++ `enum`. An additional constant is generated to guarantee that the type occupies a 32 bits wide integer.

For instance, the following IDL:

```
enum e {
    value1,
```

```

    value2
};

```

would map, according to the scoped names rules, into

```

enum e {
    value1,
    value2,
    _unused = 0xffffffff,
};

```

### 6.2.5 Mapping for strings

Gen<sup>o</sup>M IDL bounded strings are mapped to nul terminated character arrays (i.e., C strings) wrapped inside the specific `genom::bounded_string` class. Unbounded strings are mapped to `std::string` provided by the C++ standard.

For instance, the following OMG IDL declarations:

```

typedef string unbounded;
typedef string<16> bounded;

```

would map into

```

typedef std::string unbounded;
typedef genom::bounded_string<16> bounded;

```

The `genom::bounded_string` provides the following interface:

```

namespace genom3 {
    template<std::size_t L> struct bounded_string {
        char c[L];
    };
}

```

This minimalistic definition will be refined before the official 3.0 Gen<sup>o</sup>M release.

### 6.2.6 Mapping for arrays

Gen<sup>o</sup>M IDL arrays map directly to C++ arrays. All array indices run from 0 to `size-1`.

For instance, the following IDL:

```

typedef long array[4][16];

```

would map into

```

typedef int32_t array[4][16];

```

### 6.2.7 Mapping for structure types

Gen<sup>o</sup>M IDL structures map directly onto C++ `structs`. Note that these structures may potentially include padding.

For instance, the following IDL:

```

struct s {
    long a;
    long b;
};

```

would map into

```

struct s {
    int32_t a;
    int32_t b;
};

```

### 6.2.8 Mapping for union types

GenoM IDL unions map onto C structs. The discriminator in the enum is referred to as `_d`, the union itself is referred to as `_u`.

For instance, the following IDL:

```
union u switch(long) {
    case 1: long a;
    case 2: float b;
    default: char c;
};
```

would map into

```
struct u {
    int32_t _d;
    union {
        int32_t a;
        float b;
        char c;
    } _u;
};
```

Note that the C++ standard does not allow union members that have a non-trivial constructor. Consequently, the C++ mapping for such kind of unions is not allowed in GenoM either. This concerns sequences and strings, and structures or unions that contain such a type. You should thus avoid to define such datatypes in GenoM IDL in order to maximize the portability of your definitions.

### 6.2.9 Mapping for sequence types

GenoM IDL sequences mapping differ for bounded or unbounded variations of the sequence. The unbounded sequence maps onto a C++ `std::vector` provided by the C++ standard. The bounded sequences maps onto the specific `genom3::bounded_vector` class.

For instance, the following IDL:

```
typedef sequence<long> unbounded;
typedef sequence<long,16> bounded;
```

would map into

```
typedef std::vector<int32_t> unbounded;
typedef genom3::bounded_vector<int32_t, 16> bounded;
```

The `genom3::bounded_vector` provides the following interface:

```
namespace genom3 {
    template<typename T, std::size_t L> struct bounded_vector {
        T e[L];
    };
}
```

This minimalistic definition will be refined before the official 3.0 GenoM release.



# Running genom

GenoM is invoked by using one of the three following command lines:

```
genom3 [-l] [-h] [--version]
genom3 [-I dir] [-D macro[=value]] [-E|-n] [-v] [-d] file.gen
genom3 [general options] template [template options] file.gen
```

The following sections give an overview of the general behaviour (see [Section 7.1 \[Description\]](#), page 31) and detail the options affecting the GenoM program itself (see [Section 7.2 \[General options\]](#), page 31) as well as options that can be passed to templates (see [Section 7.3 \[Template options\]](#), page 33). A list of recognized environment variables is also given (see [Section 7.4 \[Environment variables\]](#), page 33).

## 7.1 Description

**genom3** generates the source code of the software components described in the formal description *file.gen* input file.

The input *file.gen* is expected to contain the description of the services, input and output ports, data types definitions and execution contexts of a software component, written in the *dotgen* language.

The dotgen specification is first processed by a C preprocessor before it is parsed by **genom3** and transformed into an abstract syntax tree. The program used as a C preprocessor can be changed with the CPP environment variable. **genom3** accepts **-I** and **-D** options that are passed unchanged to the cpp program.

The abstract syntax tree is exported in a format suitable to a *generator engine* that is in charge of a *template* execution for actual source code generation. The generator engine provides a scripting language and a set of procedures for use by templates. The directory where source code for the generator engine is searched can be changed with the **-s** option.

Templates are a set of source files that serve as the basis for source code generation. Templates source files are interpreted by the generator engine. They can contain code written in the scripting language provided by the generator engine, that computes generated output, or regular source code that is appended directly to the generated code. Intermediate files and scripts are saved in a temporary directory before they are copied to the final destination directory. The **-T** option changes the path of the temporary directory. The **-d** option will keep all temporary files instead of deleting them once the program terminates. This is useful only for template development and debugging.

The choice of a template depends on the kind of source code that is wanted by the user. Refer to the documentation of the templates for a description on what they do. The names of the available templates can be listed with the **-l** option. The directory in which templates are looked for can be changed with the **-t** option.

The **genom3** program accepts *general options* that affect the general program behaviour. **genom3** can also pass *template options* to the template. These options will only affect the template behaviour.

## 7.2 General options

**-I dir**      Add the directory *dir* to the list of directories to be searched for included files. The *dir* argument is passed as-is to the **cpp** program via the same **-I** option.

When **-r** option is in effect (either explicitly passed on the command line, or configured by default during the build process), an implicit **-I** directive pointing to the directory of the input file is appended to the end of the list of searched directories.

**-D *macro*[=*value*]**

Predefine *macro*, with definition 1 or *value* if given, in the same way as a **#define** directive would do it. This option is passed as-is to the **cpp** program.

If you are invoking **genom** from the shell, you may have to use the shell quoting character to protect shell's special characters such as spaces.

An implicit macro **\_\_GENOM\_\_** is always defined and contains the version of the **genom** program. This can be used to divert some lines in source files meant to be included by other tools that **genom**, and that contain syntax that **genom** does not understand.

**-E** Stop after the preprocessing stage, and do not run **genom** proper. The output of **cpp** is sent to the standard output. **genom** exits with a non-zero status if there are any preprocessing errors, such as a non-existent included file.

**-n****--parse-only**

Stop after the input file parsing stage, and do not invoke any template. This is useful to check the syntax of the input file. Any errors or warning are reported and **genom** exits with a non-zero status if there are errors.

**-N****--dump**

Stop after the input file parsing stage, do not invoke any template and dump the parsed specification in dotgen format. This is mostly useful for debugging **genom** itself or to view the actual specification built by **genom** from a complex (set of) file(s). Any errors or warning are reported and **genom** exits with a non-zero status if there are errors.

**-l****--list**

Print to the standard output the list of available templates. Each line of output contains the name of a template and the **genom** engine that it uses (currently, only Tcl-based templates are supported).

By default, the standard templates directory is searched, but any **-t** option will be taken into account.

**-t *path*****--tmpdir=*path***

Use *path* as the directory containing templates. This can be a colon separated list of directories which are searched in order.

This option is useful only for templates not installed in the **genom** standard directories, i.e. **share/genom/<version>/templates** or **share/genom/site-templates**.

*path* is searched for files matching **dir/\*/template.tcl**, where **\*** is the actual template name.

**-s *dir*****--sysdir=*dir***

Use *dir* as the directory holding **genom** engine files. This option is useful if non-standard engines are to be used. The default value is **share/genom/<version>/engines**.

*dir* should contain directories named after the engine name.

**-T *dir*****--tmpdir=*dir***

Use *dir* as the temporary directory holding intermediate files. See also the environment variable **TMPDIR**.

**-r****--rename**

Some **cpp** programs cannot handle correctly files with a **.gen** extension. This option will make **genom** call **cpp** with an input file ending in **.c**, linked to the real input file.

**-v**  
**--verbose** Force genom to be more verbose while processing input files.  
**-d**  
**--debug** Activate some debugging options. In particular, temporary files are not deleted. Useful for debugging genom itself or generator engines.  
**--version** Display the version number of the invoked GenoM.  
**-h**  
**--help** Print usage summary and exit.

### 7.3 Template options

**-h**  
**--help** Templates might define their own specific options. The **-h** option is always defined, and prints a summary of supported options. See the template manual for a detailed description. Template options should be passed after the template name, and before the input file name.

### 7.4 Environment variables

**CPP** Define the C preprocessor program to use. The default is `libexec/genom-pcpp`. The CPP program must recognize **-I** and **-D** arguments.

**PKG\_CONFIG** Define the path to the `pkg-config(1)` program. `pkg-config(1)` may be spawned by `genom-pcpp` for handling the `#pragma require` directive. The default is to search in the `PATH` variable.

**GENOM\_TMPL\_PATH** The value of `GENOM_TEMPLATE_PATH` is a colon-separated list of directories, much like `PATH`, where `GenoM` looks for templates. Setting this variable overrides the default search path, but any **-t** option takes precedence over this variable.

**TMPDIR** Path to the directory holding temporary files. Defaults to `/tmp`.





# Templates

Templates ...

## 8.1 The TCL backend

### 8.1.1 Runtime template information

#### Source additional template code

<code>template require <i>file</i></code>	[TCL Backend]
---	---------------

Source tcl *file* and make its content available to the template files. The file name can be absolute or relative. If it is relative, it is interpreted as relative to the template directory (pxref{dotgen template dir}).

#### Parameters:

<i>file</i>	Tcl input file to source. Any procedure that it creates is made available to the template files.
-------------	--

#### Generate template content

<code>template parse [<i>args list</i>] [<i>perm mode</i>] [<i>file string raw file ...</i>]</code>	[TCL Backend]
---	---------------

This is the main template function that parses a template source file and instantiate it, writing the result into the current template directory (or in a global variable). This procedure should be invoked for each source file that form a GenOM template.

When invoking `template parse`, the last two arguments are the destination file or string. A destination file is specified as `file file` (the filename is relative to the current template output directory). Alternatively, a destination string is specified as `string var`, where *var* is the name of a *global* variable in which the template engine will store the result of the source instantiation.

The output destination file or string is generated by the template from one or several input source. An input source is typically a source file, but it can also be a string or raw (unprocessed) text. An input source file is specified with `file file`, where *file* is a file name relative to the template directory. An input source read from a string is specified as `string text`, where *text* is the text processed by the template engine. Finally, a raw, unprocessed source that is copied verbatim to the destination is specified as `raw text`, where *text* is the text to be output.

Additionally, each input source, defined as above, can be passed a list of optional arguments by using the special `args list` construction as the *first* argument of the `template parse` command. The list given after `args` can be retrieved from within the processed template source files from the usual `argv` variable.

#### Parameters:

<i>args list</i>	This optional argument should be followed by a list of arguments to pass to the template source file. It should be the very first argument, otherwise it is ignored. Each element of the list is available from the template source file in the <code>argv</code> array.
------------------	--

*perm mode* This optional argument may be set to specify the permissions to be set for the created file.

#### Examples:

```
template parse file mysrc file mydst
```

Will parse the input file `mysrc`, process it and save the result in `mydst`.

```
template parse args {one two} file mysrc file mydst
```

Will do the same as above, but the template code in the input file `mysrc` will have the list `{one two}` accessible via the `argv` variable.

```
template parse string "test" file mydst
```

Will process the string `"test"` and save the result in `mydst`.

### Create symbolic links

<code>template link src dst</code>	[TCL Backend]
------------------------------------	---------------

Link source file *src* to destination file *dst*. If relative, the source file *src* is interpreted as relative to the template directory and *dst* is interpreted as relative to the current output directory. Absolute file name can be given to override this behaviour.

### Define template options

<code>template options { pattern body ... }</code>	[TCL Backend]
--	---------------

Define the list of supported options for the template. Argument is a Tcl switch-like script that must define all supported options. It consists of pairs of *pattern body*. If an option matching the *pattern* is passed to the template, the *body* script is evaluated. A special body specified as `"-"` means that the body for the next pattern will be used for this pattern.

#### Examples:

```
template options {
  -h - --help { puts "help option" }
}
```

This will make the template print the text `"help option"` whenever `-h` or `--help` option is passed to the template.

### Template dependencies

<code>template deps</code>	[TCL Backend]
----------------------------	---------------

Return the comprehensive list of template files processed so far. This includes files processed via `template require`, `template parse` and `template link`. This list is typically used to generate dependency information in a Makefile.

### Retrieve options passed to templates

<code>template arg</code>	[TCL Backend]
---------------------------	---------------

Return the next argument passed to the template, or raise an error if no argument remains.

## Define template help string

<code>template usage [<i>string</i>]</code>	[TCL Backend]
---	---------------

With a *string* argument, this procedure defines the template "usage" message. Unless the template redefines a `-h` option with `template options` (see [\[template options\]](#), page 36), the default behaviour of the template is to print the content of the `template usage` string when `-h` or `--help` option is passed to the template.

`template usage`, when invoked without argument, returns the last usage message defined.

## Print runtime information

<code>template message [<i>string</i>]</code>	[TCL Backend]
---	---------------

Print *string* so that it is visible to the end-user. The text is sent on the standard error channel unconditionally.

## Abort template processing

<code>template fatal [<i>string</i>]</code>	[TCL Backend]
---	---------------

Print an error message and stop. In verbose mode, print the source location as reported by the TCL command `[info frame]`.

## 8.1.2 Backend configuration

### Engine output configuration

<code>engine mode [[+]<i>modespec</i>]....</code>	[TCL Backend]
---	---------------

Configures various engine operating modes. `engine mode` can be invoked without argument to retrieve the current settings for all supported modes. The command can also be invoked with one or more mode specification to set these modes (see *modespec* argument below).

#### Parameters:

*modespec*    A mode specification string. If *mode* string is prefixed with a dash (-), it is turned off. If *mode* is prefixed with a plus (+) or not prefixed, it is turned on. Supported *modespec* are:

**overwrite**    when turned on, newly generated files will overwrite existing files without warning. When turned off, the engine will stop with an error if a newly generated file would overwrite an existing file. **overwrite** is by default off.

**move-if-change**    when turned on, an existing file with the same content as a newly generated file will not be modified (preserving the last modification timestamp). When off, files are systematically updated. **move-if-change** is on by default.

**merge-if-change**    when turned on, existing destination files will be merged with new content by the engine, instead of being overwritten (see [\[engine merge-tool\]](#), page 38). **merge-if-change** is off by default.

**debug**            when on, this mode preserves temporary files and tcl programs generated in the temporary directory. Useful only for debugging the template.

**Returns:**

When called without arguments, the command returns the current configuration of all engine modes.

**Example:**

```
engine mode -overwrite +move-if-change
```

## Automatic merge of generated content

<code>engine merge-tool tool</code>	[TCL Backend]
-------------------------------------	---------------

Changes the engine merge tool. When the engine is in 'merge-if-change' mode (see see [\[engine mode\]](#), [page 37](#)), a merge tool is invoked with the two conflicting versions of the destination file. If the merge tool exits successfully, the generated file is replaced by the merged version.

There are two builtin tools: **interactive** and **auto**. **interactive** interactively prompts the user for each patch to be applied to merge the final destination. The user can accept or reject the patch, or leave the destination file unchanged. The **auto** builtin tool automatically merges the two files and places conflict markers (<<<<<< and >>>>>>) were appropriate in the destination file.

**Parameters:**

*tool*            The path to the merge tool executable (e.g. `meld`), or one of the builtin keywords **interactive** or **auto**.

## Change output directory

<code>engine chdir dir</code>	[TCL Backend]
-------------------------------	---------------

Change the engine output directory. By default, files are generated in the current directory. This command can be used to generate output in any other directory.

**Parameters:**

*dir*            The new output directory, absolute or relative to the current working directory.

## Get current output directory

<code>engine pwd</code>	[TCL Backend]
-------------------------	---------------

**Returns:**

The current engine output directory.

### 8.1.3 Input file information

#### Genom program path and command line

Those commands implement access to genom program parameters or general information.

<code>dotgen genom program</code>	[TCL Backend]
-----------------------------------	---------------

Return the absolute path to the GenoM executable currently running.

<code>dotgen genom cmdline</code>	[TCL Backend]
-----------------------------------	---------------

Returns a string containing the options passed to the `GenoM` program.

<code>dotgen genom version</code>	[TCL Backend]
-----------------------------------	---------------

Returns the full version string of the `GenoM` program.

<code>dotgen genom templates</code>	[TCL Backend]
-------------------------------------	---------------

Return the list of all currently available templates name.

<code>dotgen genom debug</code>	[TCL Backend]
---------------------------------	---------------

Returns a boolean indicating whether `genom` was invoked in debugging mode or not.

<code>dotgen genom verbose]</code>	[TCL Backend]
------------------------------------	---------------

Returns a boolean indicating whether `genom` was invoked in verbose mode or not.

### Template path and directories

Those commands return information about the template currently being parsed.

<code>dotgen template name</code>	[TCL Backend]
-----------------------------------	---------------

Return the current template name.

<code>dotgen template dir</code>	[TCL Backend]
----------------------------------	---------------

Return a path to the template source directory (the directory holding the `template.tcl` file).

<code>dotgen template builtinidir</code>	[TCL Backend]
--	---------------

Return a path to the `genom` builtin templates source directory.

<code>dotgen template tmpdir</code>	[TCL Backend]
-------------------------------------	---------------

Return a path to the temporary directory where the template engine writes its temporary files.

### Input file name and path

Those commands return information about the current `genom` input file (`.gen` file).

<code>dotgen input notice</code>	[TCL Backend]
----------------------------------	---------------

Return the copyright notice (as text) found in the `.gen` file. This notice can actually be any text and is the content of the special comment starting with the three characters `/ * /`, near the beginning of the `.gen` file.

<code>dotgen input deps</code>	[TCL Backend]
--------------------------------	---------------

Return the comprehensive list of input files processed so far. This includes the input `.gen` file itself, plus any other file required, directly or indirectly, via a `#include` directive. This list is typically used to generate dependency information in a Makefile.

## Process additional input

<code>dotgen parse file string data</code>	[TCL Backend]
--	---------------

Parse additional `.gen` data either from a file or from a string. When parsing is successful, the corresponding objects are exported to the backend.

### Parameters:

**file|string** Specify if parsing from a file or from a string.

**data** When parsing from a file, data is the file name. When parsing from a string, data is the string to be parsed.

## Data type definitions from the specification

<code>dotgen types [pattern]</code>	[TCL Backend]
-------------------------------------	---------------

This command returns the list of type objects that are defined in the current `.gen` file. This list may be filtered with the optional *pattern* argument. Each element of the returned list is a type command that can be used to access detailed information about that particular type object.

### Parameters:

**pattern** Filter the type names with *pattern*. The filter may contain a glob-like pattern (with `*` or `?` wildcards). Only the types whose name match the pattern will be returned.

### Returns:

A list of type objects of class `type`.

## Components definitions from the specification

<code>dotgen components [pattern]</code>	[TCL Backend]
--	---------------

This command returns the list of components that are defined in the current `.gen` file. This list may be filtered with the optional *pattern* argument. Each element of the returned list is a component command that can be used to access detailed information about each particular component object.

### Parameters:

**pattern** Filter the component name. The filter may contain a glob-like pattern (with `*` or `?` wildcards). Only the components whose name match the pattern will be returned.

### Returns:

A list of component objects of class `component`.

## 8.1.4 Language dependent code generation

### Target programming language

<code>lang language</code>	[TCL Backend]
----------------------------	---------------

Set the current language for procedures that output a language dependent string

### Parameters:

**language** The language name. Must be one of `c` or `c++`.

**Canonical object name**

<code>cname <i>string</i> <i>object</i></code>	[TCL Backend]
--	---------------

Return the canonical name of the *string* or the `GenoM` *object*, according to the current language.

If a regular string is given, this procedure typically maps IDL :: scope separator into the native scope separator symbol for the current language. If a code1 object is given, this procedure returns the symbol name of the code1 for the current language.

**Parameters:**

<i>string</i>	The name to convert.
<i>object</i>	A <code>GenoM</code> object. Only class <code>code1</code> is supported at the moment.

**Generate comment strings**

<code>comment [-c] <i>text</i></code>	[TCL Backend]
---------------------------------------	---------------

Return a string that is a valid comment in the current language.

**Parameters:**

<i>c</i>	The string to use as a comment character (overriding current language).
<i>test</i>	The string to be commented.

**Canonical file extension**

<code>fileext [-<i>kind</i>]</code>	[TCL Backend]
-------------------------------------	---------------

Return the canonical file extension for the current language.

**Parameters:**

<i>kind</i>	Must be one of the strings <code>source</code> or <code>header</code> .
-------------	---

**Generate indented text**

<code>indent [#<i>n</i> ++ -] [<i>text</i> ...]</code>	[TCL Backend]
--	---------------

Output *text*, indented to the current indent level. Each *text* argument is followed by a newline. Indent level can be changed by passing an absolute level with `#n`, or incremented or decremented with `++` or `--`.

**Parameters:**

<i>test</i>	The string to output indented.
-------------	--------------------------------

**Generate filler string**

<code>--- [-<i>column</i>] <i>text</i> ... <i>filler</i></code>	[TCL Backend]
---	---------------

This command, spelled with 3 dashes (-), return a string of length *column* (70 by default), starting with *text* and filled with the last character of the *filler* string.

**Parameters:**

<i>text</i>	The text to fill.
<i>filler</i>	The filler character.
<i>column</i>	The desired length of the returned string.

**Chop blocks of text**

<code>wrap [-column] text [prefix] [sep]</code>	[TCL Backend]
---	---------------

Chop a string into lines of length *column* (70 by default), prefixed with *prefix* (empty by default). The string is split at spaces by default, or at *sep* if given.

**Parameters:**

<i>text</i>	The text to fill.
<i>prefix</i>	A string prefixed to each line.
<i>sep</i>	The separator for breaking text.
<i>column</i>	The desired maximum length of each line

**IDL type language mapping**

<code>language mapping [type]</code>	[TCL Backend]
--------------------------------------	---------------

Generate and return a string containing the mapping of *type* for the current language, or of all types if no argument is given. The returned string is a valid source code for the language.

**Parameters:**

<i>type</i>	A 'type' object.
-------------	------------------

**Code for type declarations**

<code>language declarator type [var]</code>	[TCL Backend]
---	---------------

Return the abstract declarator for *type* or for a variable *var* of that type, in the current language.

**Parameters:**

<i>type</i>	A 'type' object.
<i>var</i>	A string representing the name of a variable of type <i>type</i> .

**Code for variable addresses**

<code>language address type [var]</code>	[TCL Backend]
--	---------------

Return an expression evaluating to the address of a variable in the current language.

**Parameters:**

<i>type</i>	A 'type' object.
<i>var</i>	A string representing the name of a variable of type <i>type</i> .

**Code for dereferencing variables**

<code>language dereference type [var]</code>	[TCL Backend]
--	---------------

Return an expression dereferencing the address of a variable in the current language.

**Parameters:**

<i>type</i>	A 'type' object.
<i>var</i>	A string representing the name of a variable of type <i>type</i> .



**Code for declaring functions arguments**

<code>language argument type kind [var]</code>	[TCL Backend]
--	---------------

Return an expression that declares a parameter *var* of type *type*, passed by value or reference according to *kind*.

**Parameters:**

<i>type</i>	A 'type' object.
<i>kind</i>	Must be <b>value</b> or <b>reference</b> .
<i>var</i>	A string representing the name of a variable of type <i>type</i> .

**Code for passing functions arguments**

<code>language pass type kind [var]</code>	[TCL Backend]
--	---------------

Return an expression that passes *var* of type *type* as a parameter, by value or reference according to *kind*.

**Parameters:**

<i>type</i>	A 'type' object.
<i>kind</i>	Must be <b>value</b> or <b>reference</b> .
<i>var</i>	A string representing the name of a variable of type <i>type</i> .

**Code for accessing structure members**

<code>language member type mlist</code>	[TCL Backend]
---	---------------

Return the language construction to access a member of a *type*. *mlist* is a list interpreted as follow: if it starts with a letter, *type* should be an aggregate type (like **struct**); if it starts with a numeric digit, *type* should be an array type (like **sequence**).

**Parameters:**

<i>type</i>	A 'type' object.
<i>mlist</i>	A list of hierarchical members to access.

**Code for declaring code signatures**

<code>language signature code1 [separator] [location]</code>	[TCL Backend]
--	---------------

Return the signature of a code in the current language. If *separator* is given, it is a string that is inserted between the return type of the code and the code name (for instance, a `\n` in C so that the symbol name is guaranteed to be on the first column).

**Parameters:**

<i>code</i>	A 'code' object.
<i>separator</i>	A string, inserted between the return type and the code symbol name.
<i>location</i>	A boolean indicating whether to generate <b>#line</b> directives corresponding to the code location in <code>.gen</code> file.

**Code for calling codels**

---

<code>language invoke codel params</code>	[TCL Backend]
---	---------------

---

Return a string corresponding to the invocation of a codel in the current language.

**Parameters:**

- |               |  |
|---------------|--|
| <i>code</i>   | A 'codel' object.  |
| <i>params</i> | The list of parameters passed to the codel. Each element of this list must be a valid string in the current language corresponding to each parameter value or reference to be passed to the codel (see <a href="#">[language pass]</a> , page 43). |

# Indices

## Index of concepts

### #

#pragma.....	10
#pragma requires.....	10

### A

attribute, declaration.....	13
-----------------------------	----

### C

code1, declaration.....	14
component, declaration.....	10
Constant, declaration.....	15

### D

declaration, attribute.....	13
declaration, code1.....	14
declaration, component.....	10
declaration, ids.....	12
declaration, interface.....	12
declaration, port.....	13
declaration, service.....	13
declaration, task.....	12
dependency.....	10
dotgen.....	9
dotgen, grammar.....	17
Dotgen, identifier.....	16
dotgen, preprocessing.....	9
dotgen, specification.....	10

### G

GenoM3, grammar.....	17
GenoM3, specification.....	10
grammar.....	17

### I

identifier.....	16
ids, declaration.....	12
Input, file format.....	9
input, grammar.....	17
input, preprocessing.....	9
interface, declaration.....	12

### M

module, declaration.....	15
--------------------------	----

### P

package, dependency.....	10
parameters, service.....	14
pkg-config.....	10
PKG_CONFIG.....	10
port, declaration.....	13
preprocessing.....	9

### R

require.....	10
--------------	----

### S

service, declaration.....	13
service, parameters.....	14
specification.....	10

### T

task, declaration.....	12
Type, declaration.....	15
Type, specification.....	15

## Index of TCL backend procedures

-  
--- ..... 41

### dotgen genom verbose

dotgen genom verbose] ..... 39

## C

cname ..... 41  
comment ..... 41

## D

dotgen components ..... 40  
dotgen genom ..... 38  
dotgen genom cmdline ..... 39  
dotgen genom debug ..... 39  
dotgen genom program ..... 38  
dotgen genom templates ..... 39  
dotgen genom version ..... 39  
dotgen input ..... 39  
dotgen input deps ..... 39  
dotgen input notice ..... 39  
dotgen parse ..... 40  
dotgen template ..... 39  
dotgen template builtindir ..... 39  
dotgen template dir ..... 39  
dotgen template name ..... 39  
dotgen template tmpdir ..... 39  
dotgen types ..... 40

## E

engine chdir ..... 38  
engine merge-tool ..... 38  
engine mode ..... 37

engine pwd ..... 38

## F

fileext ..... 41

## I

indent ..... 41

## L

lang ..... 40  
language address ..... 42  
language argument ..... 43  
language declarator ..... 42  
language dereference ..... 42  
language invoke ..... 44  
language mapping ..... 42  
language member ..... 43  
language pass ..... 43  
language signature ..... 43

## T

template arg ..... 36  
template deps ..... 36  
template fatal ..... 37  
template link ..... 36  
template message ..... 37  
template options ..... 36  
template parse ..... 35  
template require ..... 35  
template usage ..... 37

## W

wrap ..... 42