

Tesseræ Reference Manual

Asher Hoskins

DRAFT

Tesseract Reference Manual

Contents

1	Introduction	3
1.1	What is Tesseract?	3
1.2	What isn't it?	3
1.3	Grammar Syntax	3
1.4	Implementation Notes	4
2	Hierarchical Organisation	4
2.1	Hierarchical Naming	4
2.2	File Usage	4
2.3	The Top-Level Component	4
3	Basic Definitions	5
3.1	Comments	5
3.2	Names	5
3.3	Numbers	5
3.4	Variable Types	7
	3.4.1 Pointers	8
	3.4.2 Floating Point Numbers	8
3.5	Type Definitions	8
	3.5.1 Structures	9
3.6	Arrays	9
	3.6.1 Enumerated Types	9
4	Interface Definitions	10
4.1	Interface Functions	10
4.2	State Interfaces	10
4.3	Parameter Interfaces	11
4.4	Extending Existing Interfaces	11
4.5	Restricting Interface Use	11
5	Configurations and Binding	11
5.1	Configuration Interfaces	12
5.2	Components in Configurations	12
	5.2.1 Nodes	13
5.3	Parameters	13
	5.3.1 Parameterised Clones	13
5.4	Binding	14
5.5	Conditional Configurations	14
	5.5.1 Pragmatic Binding	15
5.6	Fan-In and Fan-Out	15

6	Implementations	17
6.1	Setting the Implementation Language	17
6.2	Implementation Interfaces	17
6.3	Instance Variables and Constants	18
6.4	Parameters	18
6.5	Function Definitions	18
6.6	Private Functions and Prototypes	19
6.7	Inline Functions	19
6.8	Pass-Throughs	19
6.9	Ignore Functions	20
6.10	Interrupts and Static Implementations	20
6.11	The New Function	21
6.12	State Interfaces on Implementations	21
A	Example of Tesseract Code	22
A.1	The Top-Level Configuration	22
A.2	Interface Definitions	24
A.3	The Dynamic Configuration	24
A.4	Poll-Data Component	25

1 Introduction

1.1 What is Tesseract?

Tesseract is an architecture description language. It allows a program to be described in terms of “components”, each of which is a self-contained unit of code. The components are connected to each other by “interfaces”, essentially lists of function calls which must be implemented by any component that provides the interface. Interfaces on components may only be connected if they are of the same type. Connections between components may be static, fixed at compile time, or dynamic, able to change at run time. The compiler will check that all changes that could occur at run time are valid.

Components are single pieces of code, written in a number of different “source languages”, or a collection of other components and binding instructions, in which case they are called “configurations”.

After combining components together, Tesseract will output a single Aisle (intermediate code) program with as much static evaluation done and as much unused code removed as possible. This can then be passed to a suitable code generator for the target architecture.

1.2 What isn't it?

Some component description languages, such as Darwin [1], allow components to be distributed across multiple processors on a network. Tesseract, at least in its current version, does not allow this. All components must be on the same processor¹.

1.3 Grammar Syntax

The Tesseract language is described using a simple formal grammar. The grammar is made up of “symbols”, in *this font*, and “literals”, in **this font**. Symbols are defined as follows, with alternative definitions on different lines:

```
symbol:
  definition
  alternative-definition
```

Where a symbol is defined as one of a list of narrow alternatives these are shown on a single line preceded by the phrase “one of”:

```
symbol: one of
  one two three four
```

The top level symbol in a grammar is shown underlined when it is defined:

```
top-level:
  everything-else
```

Symbols that are optional have an “opt” subscript, like *this_{opt}*. A list of one or more symbols is shown like *this_{list}*. These two notations are combined to show a list of zero or more symbols, like *this_{listopt}*. Where a literal can be any one of a number of single characters in a range it is shown with a dash separating the first and last characters in the range like this: 0–9.

Some symbols, such as *ascii-character*, are left undefined. Their definition (in this case any valid ASCII character) should be obvious from their name.

¹Note that this is a limitation of this implementation, not of the language.

1.4 Implementation Notes

Implementation Note

Some of the features of Tesseract have not been implemented in the current compiler. Notes about these features, as well as thoughts on possible future extensions to the language, are displayed in a box like this.

2 Hierarchical Organisation

All interfaces, configurations and components (collectively called “entities”) used in the Tesseract system are organised into one or more hierarchical trees called “kits”. The combination of kits to be used is given at compile time. These kits are overlaid to produce a single tree called the “working kit” which is what the compiler refers to during compilation. An error will occur if an attempt is made to use an entity that is duplicated (by being in the same place in more than one kit).

2.1 Hierarchical Naming

When naming a component, interface, or variable type the separator “:” is used to separate the names of the different levels, so that `x:y` could be the component `y` in the directory `x`.

The names of interfaces on a particular component, functions on interfaces, and variable types in interfaces are separated in the same way.

```
hier-name:  
  ::opt name  
  hier-name name
```

If a name begins with “:” then it will be considered relative to the working kit directory of the current entity. For example, if the component `x:y` accesses “:z” then the entity that will be searched for will be “x:z”.

2.2 File Usage

Configurations and implementations are stored in files with the extension “.tc” and interfaces are stored in files with the extension “.ti”. There may only be a single entity per file and that entity must have the same name as the file holding it, minus the file extension.

2.3 The Top-Level Component

Compilation of a Tesseract program is started by describing the working kit and giving the name of a component to the compiler. This is referred to as the “top-level component”.

For the produced program to actually run one or more of the components in it must bind to a static instance of a component called “`sys::init`”. This provides a single `sys::init` interface called “`init`”. The component will first cause the event `init()` on this interface and then the event `run()`.

3 Basic Definitions

3.1 Comments

Anything following the characters “//” on a line is ignored.

3.2 Names

The names used to identify entities in Tesseract may be made up of alphanumeric characters and underscores providing the first character is not a number. Names are case-sensitive and there is only a single namespace for each file and so all interface, variable and variable type, and parameter names in an entity must be unique within that entity.

name:
alpha alphanumeric_{list opt}

alpha: one of
a-z A-Z _

alphanumeric: one of
alpha 0–9

Names may not be the same as any of the reserved keywords.

aisle	bind	bool	byte
case	char	clone	command
component	configuration	const	default
destroy	double	enum	event
extends	FALSE	float	ignore
implementation	inline	int	interface
interrupt	language	long	new
node	parameter	passthru	provides
requires	short	signed	state
static	struct	TRUE	typedef
unsigned	void	with	

3.3 Numbers

Integers may be expressed in binary (prefix “0b”), octal (prefix “0”), hexadecimal (prefix “0x”), or decimal (no prefix). Floating point numbers may use “e” or “E” to identify an exponential suffix. Integers may have the suffix “u” or “U” to specify that they are unsigned. Character constants consist of a printable ASCII character or character escape (as shown in table 1) enclosed in single quotes.

number:
number-sign_{opt} unsigned-number

number-sign: one of
- +

unsigned-number:
boolean-value
binary-integer
octal-integer

Name	ASCII	Value	Escape Sequence
Audible alert	BEL	7	\a
Backspace	BS	8	\b
Horizontal tab	HT	9	\t
Newline	NL	10	\n
Vertical tab	VT	11	\v
Form feed	NP	12	\f
Carriage return	CR	13	\r
Double quote	"	34	\"
Single quote	'	39	\'
Closing bracket)	41	\)
Backslash	\	92	\\
Octal code		<i>ooo</i>	\o, \oo, \ooo
Hexadecimal code		<i>hh</i>	\xh, \xhh

Table 1: Character escapes.

decimal-integer
hexadecimal-integer
character-constant
simple-float
exponential-float

boolean-value: one of
FALSE TRUE

binary-integer:
0b *binary-digit*_{list} *unsigned*_{opt}

binary-digit: one of
0 1

octal-integer:
0 *octal-digit*_{list} *unsigned*_{opt}

octal-digit:
0-7

decimal-integer:
*decimal-digit*_{list} *unsigned*_{opt}

decimal-digit:
0-9

hexadecimal-integer:
0x *hexadecimal-digit*_{list} *unsigned*_{opt}

hexadecimal-digit: one of
0-9 a-f A-F

unsigned: one of
u U

character-constant:
' *ascii-character* **'**

Type	Bytes	Notes
<code>char</code>	1	Character/8-bit integer (signed by default)
<code>byte</code>	1	8-bit integer (unsigned by default)
<code>short</code>	2	Integer (signed by default)
<code>int</code>	4	Integer (signed by default)
<code>float</code>	4	Single-precision floating point
<code>double</code>	8	Double-precision floating point

Table 2: Standard variable sizes.

simple-float:

decimal-integer . decimal-integer

exponential-float:

simple-float exponential number-sign_{opt} decimal-integer

exponential: one of

`e` `E`

Where constants are required by the language a normal number or enum name may be used.

constant:

number

hier-name

3.4 Variable Types

The sizes of the standard variable types in Tesseract are fixed², as shown in table 2, independent of the target architecture. Note that the endianness of variables *can* vary depending on the architecture.

Depending on the source language used there may be a translation between the variable type names used in Tesseract and the variable type names used in the source language, consult the source language compiler documentation for details.

variable-type:

basic-var-type

name

hier-var-type:

basic-var-type

hier-name

basic-var-type:

variable-sign_{opt} signed-types

unsigned-types

variable-sign: one of

`signed` `unsigned`

signed-types: one of

`char` `byte` `short` `int` `float` `double`

²The variable names and sizes are the same as those used on the ANS [2]

unsigned-types: one of
void bool

The `bool` type is a simple integer that is only guaranteed to be large enough to hold the values `FALSE` (equivalent to zero) and `TRUE` (equivalent to one).

3.4.1 Pointers

Pointers are not allowed in Tesseract since variable addresses can change as components are rebound. This means that any function that stores the address of a variable is likely to have that address invalidated. Pointers to functions are forbidden for the same reason³.

3.4.2 Floating Point Numbers

Floating point number support can take up a lot of resources on some of the very constrained hardware that Tesseract was designed to work on and therefore is not guaranteed to be implemented on all architectures. All integer types will always be available regardless of the target's native word size.

Implementation Note

There is currently no floating point support on the “Beastie” platform and no immediate plans to implement any.

3.5 Type Definitions

New types, defined in terms of existing types or consisting of structures or enumerated types, may be created in an interface or implementation definition.

type-defn:
typedef *variable-defn* *name* ;

variable-defn:
variable-type
structure-defn
enum-defn

A type created in one interface may not be used in another unless interface extension (see section 4.4) is used.

Tesseract has the slightly unusual feature that if a type definition name is reused then the old type definition will be silently overwritten with the new definition. Where type definitions are written in terms of other type definitions the final type definition will use whatever sub-definitions are in force at the *end* of the interface or implementation, regardless of what order they were defined in. This feature allows type definitions in extended interfaces to be overwritten and extended.

³In the current implementation code addresses are fixed, but this will change as soon as runtime loadable code is implemented.

3.5.1 Structures

Structures may be used to create composite types. Since these are always passed by value on interfaces care should be taken to ensure they are not too large⁴.

```
structure-defn:
    struct { struct-part-defnlist }

struct-part-defn:
    variable-type name arrayopt ;
```

3.6 Arrays

Attaching a list of constants in square brackets to the end of a variable name will create a fixed size array of that variable. Variable sized arrays are not permitted.

```
array:
    array-dimensionlist

array-dimension:
    [ constant ]
```

Implementation Note

Multi-dimensional arrays are not yet allowed. A future version of Tesseract may provide them.

3.6.1 Enumerated Types

Enumerated types (named signed integers) are dealt with rather more strictly in Tesseract than in C. A name from one enumerated type may not be used to set a variable of a different enumerated type, and when the value of an enumerated type name is assigned to an integer the full hierarchical name (*interface-name::enum-type::enum-name*) of the enum value required must be used. This is to avoid problems with different enumerated types in different interface definitions using the same names.

```
enum-defn:
    enum { enum-val-list }

enum-val-list:
    set-enum-val
    enum-val-list , set-enum-val

set-enum-val:
    name enum-valopt

enum-val:
    = number
```

⁴Programmers familiar with C may note that there is no name after the **struct**, this is because Tesseract does not allow pointers and so self-referential structures are not possible.

4 Interface Definitions

Interfaces define how data is to be transferred between components. They are defined separately from component definitions so that different components can share the same interface types. An interface is either a list of function calls or a single “state” or “parameter” variable definition, to be used to control dynamic components and configure parameterised components. Interfaces may be empty (for use as placeholders) or contain nothing but type definitions.

```
interface-defn:
    ifce-restrictopt interface name extend-ifcelist opt { ifce-defnopt }
```

```
ifce-defn:
    type-defnlist opt ifce-funclist opt
    type-defnlist opt state-var-defn
    type-defnlist opt param-var-defn
```

4.1 Interface Functions

All the function calls listed in an interface definition are of one of two types: **command** or **event**. A component that **provides** an interface must implement all the commands and a component that **requires** an interface must implement all the events. If a component doesn't fully implement an interface then the Tesseract compiler will refuse to use it. All function names within an interface must be unique (i.e. both commands and events share the same namespace).

```
ifce-func:
    command-event variable-type name ( func-args ) ;
```

```
command-event: one of
    command event
```

```
func-args:
    void
    func-arg-list
```

```
func-arg-list:
    func-arg
    func-arg-list , func-arg
```

```
func-arg:
    variable-type name arrayopt
```

Note that all variables are passed by value to functions and so the use of large arrays or structure types in a function definition could cause poor performance.

4.2 State Interfaces

A “state” interface can be used to control the configuration of interfaces. It contains a single statement giving the type of the variable that the state interface uses. This can be any contiguous enumerated type that starts at zero.

```
state-var-defn:
    state variable-type ;
```

In use, state interfaces have a single command, `void state()`, which is used to select one of a number of states. The state changing operation is atomic.

A configuration may have more than one state interface. The Tesseract compiler will check that all possible states of a configuration are valid.

4.3 Parameter Interfaces

Parameter interfaces are used to create parameterised clones of components. Parameters may be accessed as constants by the program code that makes up an implementation. A parameter may be any scalar, non-composite, variable type (no arrays or structures).

```
param-var-defn:  
parameter variable-type ;
```

4.4 Extending Existing Interfaces

An interface may be extended so that new functions or type definitions can be added.

```
extend-ifce:  
extends hier-name
```

Polymorphic functions are not allowed in interface definitions. If a function is defined multiple times in an interface then it is the last definition of that function, considering all `extend` statements in order followed by the interface definition body, that is used.

Since there is no requirement for an interface definition to actually contain any commands or events they may also be used as convenient places to store type definitions. If these interfaces are extended by another interface then all the types defined are available to it.

Interfaces that contain functions may only be extended by other interfaces that contain functions and state interfaces may only be extended by other state interfaces. When a state interface is extended it is the last definition of the state variable (using the order given above) that is used.

4.5 Restricting Interface Use

Interfaces can be declared as “static”, which prevents their use in a dynamic configuration. An existing unrestricted interface can be made static by defining a static interface which extends it without including any additional functions. Restrictions are inherited: it is not possible to derestrict an interface using extension.

```
ifce-restrict:  
static
```

5 Configurations and Binding

A “configuration” is a list of zero or more components and a description of how they are bound together. An interface on one component may be connected to an interface on another only if they are of the same type. The configuration itself may also have external interfaces which can be connected to its components.

```
config-defn:  
configuration name { config-statementlist }
```

config-ifce-statement:
config-statement
interface-use
parameter-use

config-statement:
component-use
binding
conditional-block

5.1 Configuration Interfaces

Configurations may have zero or more interfaces. The number and types of the interfaces on a configuration is fixed at compile time and so interfaces may not be declared inside conditional blocks.

interface-use:
interface-direction interface-type name ;

interface-direction: one of
provides **requires**

interface-type:
hier-name

5.2 Components in Configurations

By default, components are instantiated “statically”, with a single copy of the program code and a single set of instance variables for the component regardless of how many times an instantiation statement for that component type appears in a program. The **new** keyword may be used to create a “dynamic” instance of a component. There is still only one copy of the program code but multiple sets of instance variables. The **clone** keyword creates multiple copies of the program code, each with its own set of instance variables. Combined with parameter interfaces, the **clone** keyword may be used to create custom instances of components.

Care must be taken when using static instances of configurations that contain conditional sections. Only one instance of the configuration will ever be created and all users of that configuration will bind to the same state interfaces. This means that if one changes the state of the configuration then it will change for *all* of its users. To avoid this the **new** or **clone** keywords may be used when instantiating a configuration. Both will have their own copies of the state interfaces and so may be changed without affecting any other users of the same configuration: **new** configurations have only a single copy of the code that switches conditional sections, shared between all configurations of the same type, and **clone** configurations will have multiple copies of this code, one per clone.

component-use:
component *instance-type*_{opt} *component-type* *param-set*_{opt} *name ;*
node-use

instance-type:
new_{opt} **clone**_{opt}

component-type:
hier-name

Implementation Note

The current implementation of the Tesseract compiler does not support dynamic components, any use of the `new` keyword will produce an error message. Similarly, clones may only be created of implementations, not configurations.

5.2.1 Nodes

A “node” is a special form of component that may be used as a common connecting point for interfaces of the same type. It may be used, for example, to connect the output of one conditional section to the input of another. Each node has two interfaces of type *interface-type*: a provides interface “`p`” and a requires interface “`r`”. Each interface just passes data through to the other.

node-use:
`node interface-type name ;`

5.3 Parameters

If a component has parameter interfaces (see section 4.3) then that component may only be instantiated as a clone. Parameters may not be used in a configuration except to set parameters on components instantiated inside that configuration, assuming that the variable types match.

If a default value is given for a parameter then that value will be used for the parameter if no other value is provided when the component is instantiated.

Implementation Note

Parameters are currently only implemented for implementations, not configurations.

parameter-use:
`parameter parameter-type name parameter-defaultopt ;`

parameter-type:
`hier-name`

param-default:
`= constant`

5.3.1 Parameterised Clones

When a parameterised component is instantiated the `with` keyword is used to give values for its parameters. If a parameter has a default value then no value need be given when the component is instantiated. If no value is given and the parameter does not have a default value then an error will occur.

param-set:
`with { param-set-list }`

param-set-list:
`param-setting
param-set-list , param-setting`

param-setting:
`name = constant`

Implementation Note

If all the parameter interfaces on a component have default values then a future version of Tesseractæ may allow non-clone instantiations of this component as long as no parameters are set when the component is instantiated. This would allow the most common form of a parameterised component to behave as normal with the parameters needed only for more advanced use.

5.4 Binding

Links between interfaces on components are created with the `bind` command. Links must always be between a “requires” and a “provides” interface except where an interface is being bound to an interface provided or required by the configuration itself when they must be the same direction. The order of arguments in a `bind` is unimportant.

binding:

```
bind hier-name , hier-name ;
```

5.5 Conditional Configurations

When a configuration provides one or more state interfaces conditional configurations can be produced. The `state` instruction takes the name of a state interface and a list of `case` constructs, each of which contains the configuration statements to execute when the state interface is set to the listed constant values. The `default` case is used when the interface is set to an unlisted value.

There must be a single `state` instruction for every state interface and every possible case of each state interface must be catered for, either by explicitly listing all possible values or by using `default` cases. Conditional configurations may not change the interfaces of a component with a `provide` or `requires` instruction.

conditional-block:

```
state name { state-caselist }
```

state-case:

```
case default-constant-list { state-config-statementlist opt }
```

default-constant-list:

```
default-constant  
default-constant-list , default-constant
```

default-constant:

```
default  
constant
```

state-config-statement:

```
component-use  
binding
```

A dynamic configuration (one that uses state interfaces) is evaluated once at compile time, using whichever value evaluates to zero for each state interface type, and then again at run time every time any of the state interfaces change. State changes are atomic.

```

configuration dynamic {
  provides ::state_interface s;
  provides ::input_interface i;

  state s {
    case ALPHA {
      component ::type1 t1;
      bind i, t1::i;
    }
    case BETA {
      component ::type2 t2;
      bind i, t2::i;
    }
  }
}

```

Listing 1: A Dynamic Configuration.

5.5.1 Pragmatic Binding

While component instantiation instructions may appear inside conditional blocks this is just syntactic sugar, all component instantiation instructions act as though they are outside of the conditional blocks and so all components are instantiated when the program is started, not during a rebind. There is no concept of process control in Tesseract and so there is no way to destroy a component at runtime (since there is no way to know if the current thread of execution runs through any given component). Without the ability to destroy running instances there is little need for the ability to create new instances except at the very start of a program’s execution.

This could create a problem during a rebinding since every component must have all of its interfaces bound at all times. Tesseract solves this by splitting all interfaces into pairs of unidirectional interfaces (one for commands, one for events) and binding these separately. Consider the simple component in listing 1. This binds an input interface `i` to either of the components `t1` or `t2` depending upon the state of the state interface `s`.

A simplistic view of this configuration in both of its two states is shown in figure 1 but this has the problem that in each state one of the components is unbound. This is not allowed since were either of them to generate an event while unbound it would have nowhere to go.

The Tesseract compiler therefore splits each interface up into two unidirectional interfaces and binds these as shown in figure 2. It inserts “pragmatic” bindings, shown as dotted lines, so that neither component is ever completely disconnected. Note that when pragmatic binding is used the programmer must be aware that they may still receive commands or events from components that they may think they have disconnected. For this reason the compiler will always issue a warning whenever it creates a pragmatic binding.

To avoid pragmatic binding, “dummy” components must be created to absorb and ignore unwanted commands and events. These can then be bound to the interfaces of unbound components.

5.6 Fan-In and Fan-Out

A single component may have one or more of its interfaces connected to more than one component as long as the interface satisfies certain constraints. This multiple connection is known as “fan-in” when multiple `requires` interfaces are connected to a single `provides` interface, and “fan-out”

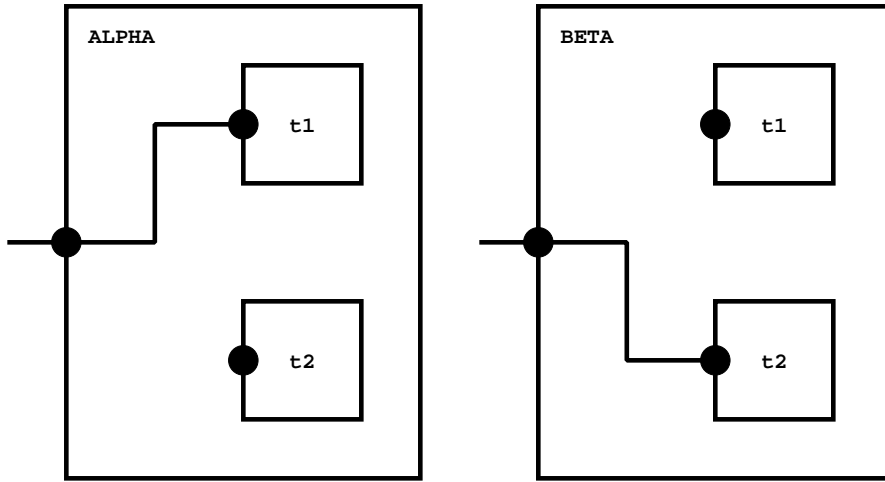


Figure 1: Simplistic Dynamic Binding.

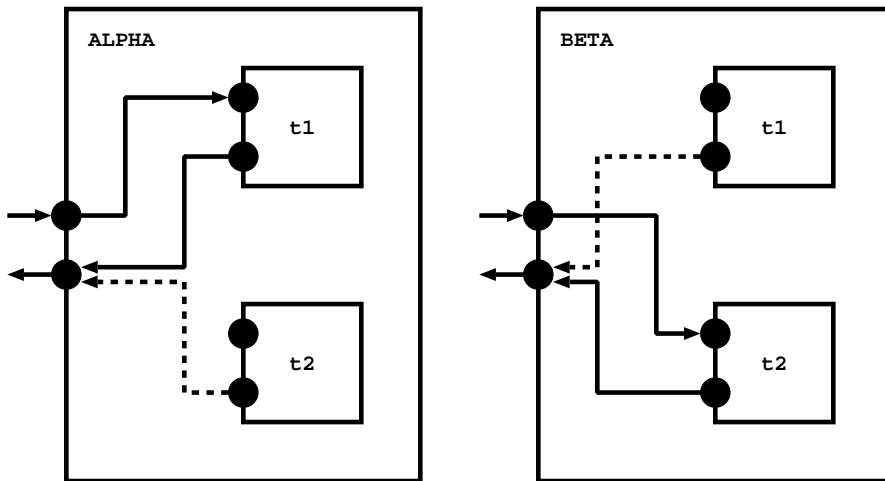


Figure 2: Pragmatic Dynamic Binding.

when a single **requires** is connected to multiple **provides**.

When a command or event is called and needs to be executed by multiple components due to fanning then these components will execute the function one after the other (this is a single threaded system). The caller of the function does not (and indeed cannot, since the program code of a particular component is completely independent of how it is bound) know whether there is one or more than one component connected to its interface and therefore is provided with a single return value. If the return type of the function is a **bool** then the return value will be **FALSE** if any of the functions called return **FALSE**. If they all return **TRUE** then the return value will be **TRUE**. If the return type of the function is anything other than **void** or **bool** then a combined return type would not make sense and so the fanned in or out binding will be prevented by the Tesseract compiler.

Only events are affected by these restrictions on a fanned-in interface, and only commands are affected on a fanned-out interface.

6 Implementations

An “implementation” is where the active parts of a component are described. Implementations can be written in a number of different languages depending on which compilers are available.

implementation-defn:

```
static-implopt implementation name { implementation-statementlist }
```

implementation-statement:

```
type-defn  
language-default  
interface-use  
parameter-use  
impl-var-defn  
constant-defn  
impl-func-defn
```

6.1 Setting the Implementation Language

The **language** statement sets the language that all function definitions following it are written in. There is no default value so an attempt to define a function without first setting a language will produce a compilation error.

language-default:

```
language-set ;
```

language-set:

```
language language-name
```

language-name: one of

```
aisle name
```

The language name “**aisle**” (the intermediate language) will always be valid. The validity of other names depends on the configuration of the compiler.

6.2 Implementation Interfaces

The interfaces that an implementation has are listed in the same way as for configurations (see section 5.1).

6.3 Instance Variables and Constants

Any variable declared outside of the functions in an implementation is added to the instance variable block. There will be one of these blocks for every instance of the component that is created.

```
impl-var-defn:
  constopt hier-var-type name constant-initopt ;

constant-init:
  = constant
  array = { constant-list }
  [] = { constant-list }

constant-list:
  constant
  constant-list , constant
```

When a variable array is initialised the number of elements in the *constant-list* must match the number of elements in the array. The shorthand “[]” may be used to set the size of the array automatically. Variables with initial values will be set to those values every time an instance of the component that contains them is created.

If the keyword “**const**” is added to the beginning of a variable definition then a constant is produced instead. Scalar (non-array) constants will take up no space in the final executable and array constants will only take up space if they are used (unused constants will get removed by the optimiser).

Implementation Note

The initialisation of arrays or structures is not currently implemented and so array or structure constants may not be used.

6.4 Parameters

Parameters on an implementation are listed in the same way as for configurations (see section 5.3). Parameters are visible as constants to the code of an implementation.

6.5 Function Definitions

Tesseract is not a programming language and so the bodies of function definitions have to be written in a different language. Tesseract actually only understands how to manipulate the intermediate language, Aisle, and so will use the **language** statement to decide which external compiler to send the function body off to for compilation into Aisle. Aisle is described in a separate manual.

```
impl-func-defn:
  func-type hier-var-type hier-name ( hier-func-argsopt ) ;
  func-type hier-var-type hier-name ( hier-func-argsopt ) func-body
  passthru
  ignore
  interrupt name ( ) func-body
  new ( ) func-body
```

```

func-type:
    inlineopt command-eventopt

hier-func-args:
    void
    hier-func-arg-list

hier-func-arg-list:
    hier-func-arg
    hier-func-arg-list , hier-func-arg

hier-func-arg:
    hier-var-type name arrayopt

func-body:
    language-setopt { source-statementlist opt }

```

The `language` tag can be added to a function definition to change the language for one function only without affecting the language settings for the rest of the implementation.

6.6 Private Functions and Prototypes

As well as functions listed in interface definitions, an implementation may contain “private” functions which are accessible only to other functions inside the implementation. These functions do not have a `command` or `event` tag and must have a function prototype (where the function body is replaced by a semicolon) listed before their main definition.

The return and argument variable types (but not names) of all functions must match those of the function prototype for both private functions and those defined in interface definitions.

Implementation Note

The variable type returned by a function must be a type with a single value. Functions may not return structs.

6.7 Inline Functions

If a function is marked as `inline` then the Tesseract compiler will inline the code it contains whenever that function is called. Functions marked as `inline` in components which are bound to dynamically will not be inlined, even if the function is being called from a component which has a static binding to its implementation component.

Inline functions may not modify their argument variables (since these will be replaced by references to variables in the calling function when inlined). The source language compiler must enforce this restriction.

The `inline` directive is only necessary on the actual definition of a function. It will be ignored on function prototypes.

6.8 Pass-Throughs

When designing a component that upgrades another component by wrapping it inside itself, perhaps with an extended interface, it is common for part of the wrapper to be an implementation that modifies the behaviour of some commands and event and just “passes through” other calls from one interface to another.

To simplify this process the `passthru` statement may be used to call a function on one interface with the arguments from a function on another. There is no need for both interfaces to be the same type, they just have to share the same function prototypes. Using a `passthru` is exactly equivalent to writing an inline function that consists of a single function call instruction.

```
passthru:
    passthru command-event in-function , out-function ;

in-function:
    hier-name ( )

out-function:
    hier-name ( )
```

6.9 Ignore Functions

There are times when a component may choose not to implement certain functions, particularly events, on interfaces. For example, all top level implementations must require the `sys::init` interface but in many cases the `init` event on that interface is not required. The `ignore` keyword is a shorthand for writing the blank inline function that is required in these cases.

```
ignore:
    ignore command-event hier-name ( ) ;
```

6.10 Interrupts and Static Implementations

Functions of type `interrupt` are interrupt entry points. They save the processor state on entry and restore it on exit. If the target processor has a special return opcode to use on exit from an interrupt routine then that will be used when the function exits. The name of the interrupt function will depend on the particular system. If multiple components use the same interrupt then the interrupt functions will be called in order. Native code must be used to enable and disable interrupts.

Interrupt functions may only appear in components that are statically instantiated. The compiler will refuse to compile dynamic interrupt functions. This is because on some systems interrupt functions are called directly by the interrupt hardware and that hardware knows nothing about Tesseract instance variable blocks and so cannot set the pointer to them.

The `static` keyword may be used to mark an implementation that can only be instantiated statically. This keyword may also be useful when writing certain types of native code that rely on an implementation being static (so that they can access the instance variables using the correct addressing mode, for example).

```
static-impl:
    static
```

When an interrupt function is called a new thread of execution is started. Further interrupts will be disabled within this thread until the first non-inline function call out of the interrupt routine. It should be born in mind that the rest of the program is blocked until the interrupt function makes this first call.

Note that the only way that an interrupt function can be entered is by an interrupt. It is not possible to jump to them as though they were normal functions since they use a special “return from interrupt” instruction to exit which will produce incorrect results if they were entered using a normal call instruction. Compilers should enforce this restriction.

Implementation Note

There really ought to be a better way to control the restarting of interrupts from within an interrupt routine. Doing it at the first call is convenient (since the stack frame manipulations needed to perform a function call require interrupts to be turned off and on around them) but could have unintended consequences in complex interrupt routines.

6.11 The New Function

If a component contains a function called `new()` then it will be run when the component is instantiated, at the very start of the program. There is no way to determine the order in which `new()` functions are run and so this function must not make any calls on interfaces since there is no way to know if the components that it is calling have been initialised yet. The `new()` function may call other functions in the component as long as they do not make interface calls either.

6.12 State Interfaces on Implementations

If a state interface is provided by an implementation then the value of that interface is available to all function definitions in the implementation as a variable with the same name as the interface. The interface will also have a single command called “`state()`” with a single argument of the same type as the state interface which may be called in exactly the same way as the `state()` function on a dynamic configuration. Note that the state variable value is not automatically changed when the state interface function is called.

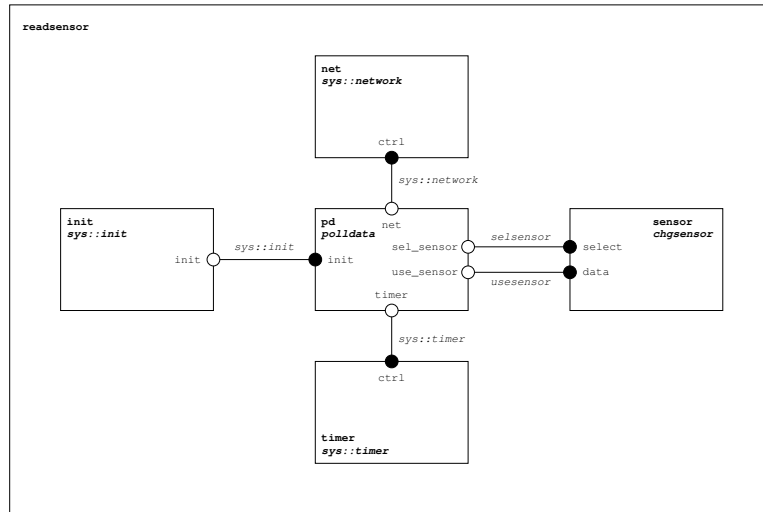


Figure 3: Example configuration: top level.

A Example of Tesseract Code

This simple example shows the configurations, interfaces, and some of the implementations required to produce the dynamic configuration shown in figures 3 and 4. This simple application polls either a heat or a light sensor every ten seconds and dumps the results out via a streaming network connection.

A.1 The Top-Level Configuration

```

1 // readsensor.tc
2 // Top-level configuration for Tesseract Example.
3
4 configuration readsensor {

```

All programs must have an instance of `sys::init` or they will never get run.

```

5     component sys::init init;

```

Now we specify what other components to use. All these components are static, so they will only be created if a component of the type requested has not already been created.

```

6     component sys::timer timer;
7     component sys::network net;
8     component ::polldata pd;
9     component ::chgsensor sensor;

```

Binding is straightforward. There's no required order for the arguments in a `bind` command so just use what seems clearest.

```

10    bind init::init, pd::init;
11    bind pd::net, net::ctrl;
12    bind pd::timer, timer::ctrl;
13    bind pd::sel_sensor, sensor::select;

```

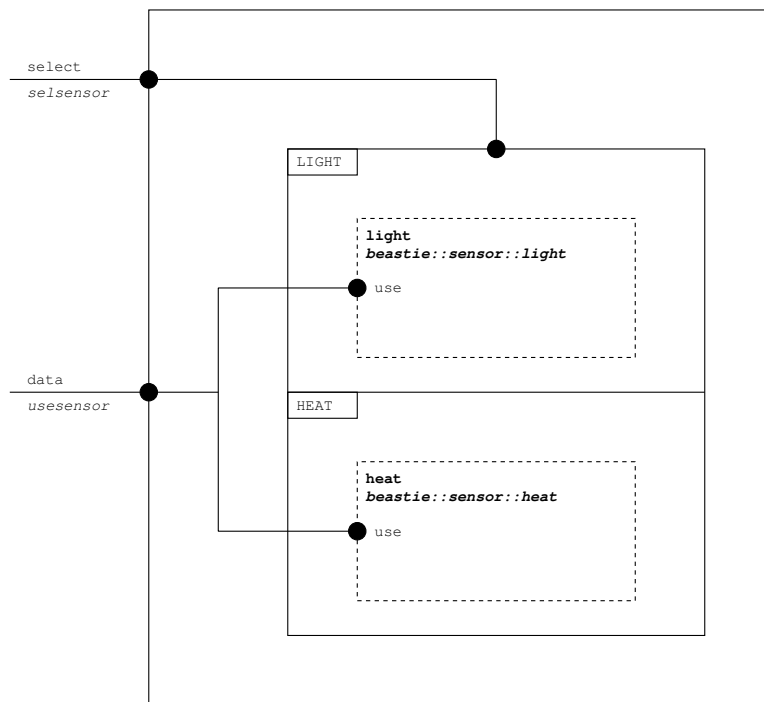


Figure 4: Example configuration: dynamic sensor selection.

```

14     bind pd::sensor_data, sensor::data;
15 }

```

A.2 Interface Definitions

Before we can define our dynamic configuration we need to create a couple of interfaces.

```

1 // selsensor.ti
2 // A state interface to select a sensor to use.
3
4 interface selsensor {

```

This interface can take either of the values “HEAT” or “LIGHT”.

```

5     typedef enum { LIGHT, HEAT } sensortype;
6     state sensortype;
7 }

```

Our sensors are very simple to use. There’s just a single command that returns an integer representing the sensor value when it’s called. This interface is part of our example’s `beastie::sensor` kit.

```

1 // usesensor.ti
2 // An interface to allow a sensor to be polled.
3
4 interface usesensor {
5     command int poll(void);
6 }

```

A.3 The Dynamic Configuration

This dynamic configuration uses a light sensor when `select` is set to `LIGHT` and a heat sensor when it’s set to `HEAT`.

```

1 // chgsensor.tc
2 // A conditional configuration that selects between two sensors.
3
4 configuration chgsensor {
5     provides ::selsensor select;
6     provides beastie::sensor::usesensor data;
7
8     state select {
9         case LIGHT {
10             component beastie::sensor::light light;
11             bind light::use, data;
12         }
13         case HEAT {
14             component beastie::sensor::heat heat;
15             bind heat::use, data;
16         }
17     }
18 }

```

A.4 Poll-Data Component

For this example the implementation of one component only is shown. The `polldata` component is written in C.

```
1 // Poll a sensor and push data out to a network.
2
3 implementation polldata {
4
5     requires sys::init init;
6     requires sys::network net;
7     requires sys::timer timer;
8     requires ::selsensor sel_sensor;
9     requires beastie::sensor::usesensor sensor_data;
10
11     language c;
12
13     // Component instance variables.
14     byte count;
15     sel_sensor::sensortype current;
16
17     // Prototype for private function.
18     void makesensor(sel_sensor::sensortype s);
19
20     // We don't have any initialisation code.
21     inline event void init::init(void) {
22     }
23
24     // Called to start running.
25     event void init::run(void) {
26         // Clear the counter and select a sensor type to start with.
27         count = 0;
28         makesensor(LIGHT);
29         // We set the network to accept a stream of data to a fixed network
30         // address.
31         net::config(STREAM, 10,0,0,1);
32         // Cause a timer event every 1000ms.
33         timer::repeat(1000);
34     }
35
36     // Whenever we receive a timer event we poll a sensor and dump the data
37     // to the network stream. Every 10 events we swap sensors.
38     event void timer::trigger(void) {
39
40         net::stream(sensor_data::poll());
41
42         count++;
43         if (count == 10) {
44             count = 0;
45             if (current == LIGHT) {
46                 makesensor(HEAT);
47             }
48             else {
49                 makesensor(LIGHT);
```

```
50         }
51     }
52
53 }
54
55 // Select the sensor to use.
56 // Note that this is a private function.
57 void makesensor(sel_sensor::sensortype s) {
58     current = s;
59     sel_sensor::state(s);
60 }
61
62 }
```

References

- [1] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
- [2] Julie A. McCann, Asher Hoskins, and Gawesh Jawaheer. ANS (autonomic networked system) for ubiquitous computing. In *UbiComp 2004 Adjunct Proceedings: Posters*, Berlin, Germany, September 2004. Springer. Extended abstract for poster.

Index

clone, *see* component, clone

component

clone, 12

dynamic, 12

static, 12

grammar

syntax of, 3

grammar symbols

alpha, 5

alphanumeric, 5

array, 9

array-dimension, 9

basic-var-type, 7

binary-digit, 6

binary-integer, 6

binding, 14

boolean-value, 6

character-constant, 6

command-event, 10

component-type, 12

component-use, 12

conditional-block, 14

config-defn, 11

config-ifce-statement, 11

config-statement, 12

constant, 7

constant-init, 18

constant-list, 18

decimal-digit, 6

decimal-integer, 6

default-constant, 14

default-constant-list, 14

enum-defn, 9

enum-val, 9

enum-val-list, 9

exponential, 7

exponential-float, 7

extend-ifce, 11

func-arg, 10

func-arg-list, 10

func-args, 10

func-body, 19

func-type, 18

hexadecimal-digit, 6

hexadecimal-integer, 6

hier-func-arg, 19

hier-func-arg-list, 19

hier-func-args, 19

hier-name, 4

hier-var-type, 7

ifce-defn, 10

ifce-func, 10

ifce-restrict, 11

ignore, 20

impl-func-defn, 18

impl-var-defn, 18

implementation-defn, 17

implementation-statement, 17

in-function, 20

instance-type, 12

interface-defn, 10

interface-direction, 12

interface-type, 12

interface-use, 12

language-default, 17

language-name, 17

language-set, 17

name, 5

node-use, 13

number, 5

number-sign, 5

octal-digit, 6

octal-integer, 6

out-function, 20

param-default, 13

param-set, 13

param-set-list, 13

param-setting, 13

param-var-defn, 11

parameter-type, 13

parameter-use, 13

passthru, 20

set-enum-val, 9

signed-types, 7

simple-float, 6

state-case, 14

state-config-statement, 14

state-var-defn, 10

static-impl, 20

struct-part-defn, 9

structure-defn, 9

- type-defn*, 8
- unsigned*, 6
- unsigned-number*, 5
- unsigned-types*, 7
- variable-defn*, 8
- variable-sign*, 7
- variable-type*, 7

implementation notes

- array/struct initialisation, 18
- arrays, 9
- component instantiation, 13
- floating point, 8
- in this manual, 4
- interrupts, 21
- parameters, 14
- parameters on configurations, 13
- return type, 19

keywords

- aisle*, 17
- bind*, 14
- bool*, 8
- byte*, 7
- case*, 14
- char*, 7
- clone*, 12
- command*, 10
- component*, 12
- configuration*, 11
- const*, 18
- default*, 14
- double*, 7
- enum*, 9
- event*, 10
- extends*, 11
- FALSE*, 6
- float*, 7
- ignore*, 20
- implementation*, 17
- inline*, 19
- int*, 7
- interface*, 10
- interrupt*, 18
- language*, 17
- new*, 12, 18
- node*, 13
- parameter*, 11, 13
- passthru*, 20
- provides*, 12
- requires*, 12
- short*, 7
- signed*, 7
- state*, 10, 14
- static*, 11, 20
- struct*, 9
- TRUE*, 6
- typedef*, 8
- unsigned*, 7
- void*, 8, 10, 19
- with*, 13