T U M

I N S T I T U T F Ü R I N F O R M A T I K

Zerberus Language Specification Version 1.0

Christian Buckl
buckl@in.tum.de

TUM-I0501
Januar 05

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

Contents

1 Introduction

Introduction and general annotations

This specification describes the syntax and semantics of the Zerberus language. The Zerberus language is part of the Zerberus system, a development environment for secure and reliable real-time control system. This specification is not intended as an introduction to the Zerberus system or its logical concept. It should rather be used as reference book for developers to implement a control system with the Zerberus System.

The specification denotes the syntax of the Zerberus language in the style of the Extended Backus-Naur form (EBNF) with some small deviations to support the readability. Some basic definitions can be found in the appendix. For illustration purpose examples are attached to each section.

Besides the pure syntax common some common implementation errors due to the semantics are described. These annotations should help the developer to write correct Zerberus code. In case of the existence of semantic errors these errors are detected by the code generator of the Zerberus system by performing tests. If semantic errors can not be excluded, warnings are generated to point out the hazards to the developer. Which errors are automatically detected by the code generator and in which cases warnings are generated is described within this specification.

If some extensions are planned for future versions these extensions are indicated.

2 General attributes

General information about the Zerberus language

This section describes the general attributes of the Zerberus language. The syntax of the Zerberus language is orientated on the syntax of common programming languages in particular on the programming language C++. Thereby the acquisition of the Zerberus language is simplified.

A Zerberus language source file for an application consists of several object declarations. Each object declaration is comprised of the object type, the object identifier and the declaration block. Within the declaration block the attributes of the related object can be determined. Comments can be used to clarify the code.

2.1 Object types

The Zerberus language defines several object types: ports (section 3), actors (section 4), sensors (section 5), guards (section 6), tasks (section 7), modes (section 8) and modechanges (section 9). These object types and their declaration syntax are explained in detail in the denoted sections.

2.2 Object identifier

The naming conventions for the different objects follow the rules of the programming languages C++. A valid identifier is a sequence of at least one letter followed by more letters, digits or underline symbols. Neither spaces nor special characters can be part of an identifier. The length of an identifier is not limited.

Keywords, see appendix B, of the Zerberus language and keywords of the used run-time programming language are reserved and can not be used as identifiers. If nevertheless keywords of the Zerberus language are used as identifier the Zerberus code generator will reject the Zerberus source file with an error. An offense against the prohibition to use keywords of the run-time programming language can not be detected by the Zerberus code generator since the Zerberus code generator is designed platform independant. The violation of this rule will be probably not detected until the platform dependant code generation process.

In future versions the Zerberus code generator will also support key word violation tests for the most common programming languages.

Within the application the identifier of an object has to be unique.

2.3 Declaration block

The declaration block of an object consists of several members. The available members are dependant on the object type and defined in this specification. If a member is optional it is stated explicitly. The order of the members within a declaration block is not binding and can be arbitrarily chosen.

2.4 Functions

In several cases the application developer has to specify functions that perform a certain job in the run-time system. One example would be the function for a task. Since this function performs the pure control computation of this task, the code of this function is application dependant. It is the job of the developer to implement the function.

The functions must be implemented platform dependant. If several platforms are used the function must be implemented several times (N-Version programming).

The implemented functions are assigned in the Zerberus code file to the objects by the function names. Since the exact signature of these functions is dependant on the platform and in particular on the used programming language, the signature is not specified in the Zerberus code file. Instead the function name is marked as function by adding a left and a right parenthesis at the end of the function name. The arguments of the functions arise out of the declaration context.

During the application development process the code generator can display the required functions with the function type and the signature for a certain run-time system.

2.5 Comments

The syntax for comments is equal to the syntax in the programming language C++. A one-line comment is prefaced by two preceding slashes. A block comment is started by `/*` characters and stopped by `*/`.

3 Ports

Unique memory spaces

A port identifies a unique space in memory. Ports are determined by their type, the identifier, the initial value and the comparison state. The value of the port can be read by actors, guards, tasks and modechanges. Sensors and tasks can write a port. To avoid race conditions and a nondeterministic execution the simultaneous write access on a port by tasks or sensors is forbidden.

Since ports are the base for the voting algorithms the size and representation of the port values is well-defined. All run-time systems must be in conformance with this determination. A run-time system must not support all port types. Therefore the code generator performs tests for a combination of application and run-time system to confirm the compatibility of this combination. If the run-time system does not support every port type used by the application, the combination of application and run-time system is rejected.

3.1 Syntax of port declaration

```
<PortDeclaration> = "port" <PortName>
                    "{"
                        "type =" <PortType> [ArraySize]";"
                        "compareMode =" <CompareMode>";"
                        "initialValue =" <InitialValue>";"
                    "}";

<PortName> = <Identifier>;
<PortType> = "CHAR" | "UCHAR" | "BOOL" | "INT16" | "INT32"
             | "INT64" | "UNIT16" | "UINT32" | "UINT64"
             | "FLOAT32" | "FLOAT64";
<ArraySize> = "["<Number>"]";
<CompareMode> = "BINARY" | "NONE" | <Function>;
<InitialValue> = ["-"]<Number>
```

```
    |["-"]<number>.<number>
                |<Letter>|<Function>;
```

3.2 Declaration members

3.2.1 Name

This member assigns a name to the port. The name has to be unique within the application and needs to follow the rules for identifiers, see 2.2.

3.2.2 Type

The type assignment declares the type of the port.

Allowed types are the different basic types:

- CHAR
- UCHAR
- BOOL
- INT16
- INT32
- INT64
- UINT16
- UINT32
- UINT64
- FLOAT32
- FLOAT64

CHAR Ports of type CHAR have a size of one byte. The content can be used to present a character or an integer. The representation of a number is in two's complement.

UCHAR Ports of type UCHAR have a size of one byte and can be used to present characters or positive integers from 0 to 255.

BOOL The BOOL type is used for ports that express a value false or true. Although one bit would be enough to represent the different values, a memory area of size of one byte is used to avoid conflicts with several programming languages. Every value different from zero will be therefore interpreted as true, a value of zero as false.

INT16 Ports of type INT16 represent memory areas with a size of two bytes. Corresponding ports can be used to express integer values from -2^{15} to $+2^{15}-1$. The representation of these numbers is in two's complement.

INT32 Ports of type INT32 represent memory areas with a size of four bytes. Corresponding ports can be used to express integer values from -2^{31} to $+2^{31}-1$ The representation of these number is in two's complement.

INT64 Ports of type INT64 represent memory areas with a size of eight bytes. Corresponding ports can be used to express integer values from -2^{63} to $+2^{63}-1$ The representation of these number is in two's complement.

UINT16 Ports of type UINT16 represent memory areas with a size of two bytes. Corresponding ports can be used to express integer values from 0 to $+2^{16}-1$.

UINT32 Ports of type UINT32 represent memory areas with a size of four bytes. Corresponding ports can be used to express integer values from 0 to $+2^{32}-1$.

UINT64 Ports of type UINT64 represent memory areas with a size of eight bytes. Corresponding ports can be used to express integer values from 0 to $+2^{64}-1$.

FLOAT32 Ports of type FLOAT32 represent memory areas with a size of four bytes. Corresponding ports can be used to express floating point numbers according to the IEEE 754 single precision standard.

FLOAT64 Ports of type FLOAT64 represent memory areas with a size of eight bytes. Corresponding ports can be used to express floating point numbers according to the IEEE 754 double precision standard.

A port can also be an array of the denoted types. For an array declaration the type is followed by the array size encircled by square brackets.

3.2.3 Compare state

There are different possibilities how a port can be treated by the system in the context of the voting. The port can have the following compare states:

- NONE
- BINARY
- Function

NONE If the compare mode of a port is NONE, the ports value is not evaluated in the voting algorithm. This comparison mode can only be applied for ports that are not read by actors or by a modechange. The first condition assures that incorrect port values can not be used for the system output; the second condition assures that the Zerberus units can not reach different modes. If such a case the temporal conditions for the voting could differ and this would result in the assumption of the failsafe condition by at least one unit.

BINARY A port with compare mode binary is evaluated in the voting algorithm every time when a task can potentially write the port or if the value is read by an actor or modechange. The comparison of the values of the different Zerberus units occurs bit-by-bit, except for ports with port type BOOL. In the latter case the value of the port is interpreted as boolean value and only these boolean values are compared.

Function Instead of the keywords NONE and BINARY a function can be determined that performs the voting. In this case the port values are used for the voting every time the port is potentially written by a port or is read by an actor or modechange. The comparison is performed by the function to allow also interval decisions. This decision can be useful if the correct values are situated in an interval. There are several possible reasons for this property. The port can be of type float, the values can be the result of non-deterministic N-Version programming or the results of task computations can be based on sensor results that differ a little bit because of the allowed temporal synchronization difference between the units.

3.2.4 Initial value

To guarantee a consistent state of all Zerberus units at system start the ports of the different units must contain the same values. To ensure this condition the ports have to be initialized in a consistent way by the application developer. The initial value can be stated directly in the Zerberus code file or an initialization function can be used. The latter possibility is especially useful for ports that represent large arrays. To use an initialization function the developer has only to state the function name followed by an opening and a closing bracket.

In case of the direct initialization the developer has to mind that the initialization must not contain any blanks.

3.3 Errors

3.3.1 Incomplete declaration

The code generator will produce an error, if one or more members of the port declarations are missing.

3.3.2 Wrong compare state

Not every port can be assigned as a non-compare port. If a port is read by an actor or by a modechange it has to be compared by the system to guarantee a secure execution of the application. If this condition is violated, the code generator will raise an error and quit the code generation process.

3.3.3 No references

If a port is not read by any actor, task, guard or modechange the code generator will stop its execution and raise a failure. The same happens if a port is not written by any sensor or task.

3.4 Examples

```
port port1
{
    type=INT32[10];
    compareMode=NONE;
    initialValue=initPort1_3();
}

port port2
{
    type=FLOAT32;
    compareMode=comparePort2();
    initialValue=0;
}

port port3
{
    type=INT32[10];
    compareMode=BINARY;
    initialValue=initPort1_3();
}
```

The first declaration specifies the port port1. This port is an integer (four bytes) array of size 10. It is not compared (thus it is not allowed that this port is read by an actor or a modechange) and is initialized by the function initPort1_3().
The second declaration introduces the port 2. The port type is float with a size of 4 bytes. As a bit by bit comparison is not eligible for a floating point number the port is compared by a function, which is implemented by the application developer, see section 10.
The last port is again an integer (four bytes) array of size 10. Its value is compared with the value of the according ports on the other Zerberus machines bit by bit. For the initialization the same function as for port1 is used.

4 Actors

Manipulating the environment

An actor identifies a frequently called function that manipulates the environment on the base of some port values. Several common actor functions are supported by the Zerberus run-time systems but it is also possible to specify a new actor function.

4.1 Syntax of actor declaration

```
<ActorDeclaration>= "actor " <ActorName>
                    "{"
                        "function =" <Function>";"
                        "in =" <PortList>";"
                    "}";

<ActorName> = <Identifier>;
<PortList> = <PortName> {","<PortName>};
```

4.2 Declaration members

4.2.1 Name

This member assigns a name to the actor. The name has to be unique within the application and needs to follow the rules for identifiers, see 2.2.

4.2.2 Actor function

The actor function specifies the function that should be executed every time the actor is called. The application developer has to state the name of the function followed by a left and a right parenthesis. The supported actor functions are described in the documentation of the specific run-time system. If the desired actor function is not available the application developer can implement his own function.

The parameters of the actor functions are the value of the ports as specified in the port list. The exact signature of the function is platform dependant and can therefore not be stated in advance. For a given combination of application and run-time system the Zerberus code generator can create a list of the actor functions with their exact signature that need to be implemented by the developer.

4.2.3 Port list

The actor function controls the environment on the base of port values. At least one port has to be assigned to the actor. Each port that is associated with an actor must be compared during the voting algorithm execution. The order of the particular ports is reflected in the order of the function parameters.

Every port in the port list has to be declared before the actor declaration.

4.3 Errors

4.3.1 Incomplete declaration

The code generator will raise an error if a member is missing in the actor declaration.

4.3.2 No references

If the actor is not used in any mode, the code generator will raise a failure message and stop its execution.

4.3.3 Invalid port compare state

Every port that is read by an actor must be compared during the voting algorithm execution. If this condition is not satisfied the Zerberus code generator will raise a failure message and stop its execution.

4.4 Examples

```
actor actor1
{
    function=myActorFunction();
    in=port2,port3;
}
```

This declaration declares an actor with the function myActorFunction(). The actor reads the ports port2 and port3.

5 Sensors

Measuring the environment

A sensor is a frequently called function that reads values from the environment and writes these values into some ports. Several common sensor functions are supported by the Zerberus run-time systems, but it is also possible to use an own function if the desired sensor function is not available.

5.1 Syntax of sensor declaration

```
<SensorDeclaration>  = "sensor " <SensorName>
                       "{"
                         "function =" <Function>";"
                         "out =" <PortList>";"
                       "}";

<SensorName> = <Identifier>;
```

5.2 Declaration members

5.2.1 Name

This member assigns a name to the sensor. The name has to be unique within the application and needs to follow the rules for identifiers, see 2.2.

5.2.2 Sensor function

The sensor function specifies the name of the function that should be executed when the sensor is called. This function name can be the name of a sensor function supported by the run-time system or of a function implemented by the application developer.

The parameters of the sensor function are the ports specified in the port list. The exact signature of the function is platform dependant and can therefore not be stated in advance. For a given combination of application and run-time system the Zerberus code generator can create a list of the sensor functions and their signatures that have to be implemented by the developer.

5.2.3 Port list

The port list specifies the ports that are written by the sensor. The order of the particular ports is reflected in the order of the function parameters.

Every port in the port list has to be declared before the sensor declaration.

5.3 Errors

5.3.1 Incomplete declaration

The code generator will produce an error if a member is missing in the sensor declaration.

5.3.2 No references

If the sensor is not used in any mode, the code generator will raise a failure message and stop its execution.

5.4 Examples

```
sensor sensor1
{
    function=mySensorFunction();
    out=port1;
}
```

This example declares a sensor that uses the function mySensorFunction and writes the result into port port1.

6 Guards

Controlling the task execution

Guards control the execution of the assigned tasks. Before every logic start of the according task the guard function evaluates on the base of port values if the task should be started or not. Currently the guard function has to be implemented by the application developer. In the future it is envisaged to specify the guard decision directly in the development tool.

6.1 Syntax of guard declaration

```
<GuardDeclaration>  = "guard "<GuardName>
                      "{"
                         "function =" <Function>";"
                         "in =" <PortList>";"
                      "}";

<GuardName> = <Identifier>;
```

6.2 Declaration members

6.2.1 Name

This member assigns a name to the guard. The name has to be unique within the application and must follow the rules for identifiers, see 2.2.

6.2.2 Guard function

The guard function member determines the name of the function that realizes the decision if a task should be executed or not. The function has to be implemented by the developer.

In the guard declaration the developer must assign the name of the guard function followed by a left and a right parenthesis.

The exact signature of the guard function depends on the platform and the ports as specified in the port list, which represent the arguments of the function. For a given combination of application and run-time system the Zerberus code generator can specify the exact signatures of the guard functions.

6.2.3 Port list

The port list specifies the ports that are read by the guard function and serve as base for the guard decision. The order of the particular ports is reflected in the order of the function parameters.

Every port in the port list has to be declared before the guard declaration.

6.3 Errors

6.3.1 Incomplete declaration

The code generator will produce an error if a member is missing in the guard declaration.

6.3.2 No references

If the guard is not used by any task, the code generator will reject the Zerberus code file.

6.4 Example

```
guard guard1
{
    function=myGuardFunction();
    in=port1;
}
```

This example declares a guard with a evaluation function called myGuardFunction. The result of the evaluation is based on the value of port1.

7 Tasks

Performing control computations

Tasks realize the real control functionality of the system. A task is thereby a frequently called pure computation function. The parameters of these functions are a subset of the ports, the result of the value is written into another subset of ports.

The execution of a task can be controlled by modes and guards. Basically a task is executed n-times in a mode cycle if the task is member of the mode with a frequency of n. At every logical start a guard can determine on the base of assigned port values if the task should be executed in this iteration.

7.1 Syntax of task declaration

```
<task_declaration>= "task " <TaskName>
                    "{"
                        "function =" <Function>";"
                        <TaskPorts>
                        ["guard =" <GuardName>";"]
                    "}";

<TaskName> = <Identifier>;
<TaskPorts> = "in =" <PortList>;
              ["inout =" <PortList>";"]
              "out =" <PortList>";"
              |
              ["in =" <PortList>";"]
              "inout =" <PortList>";"
              ["out =" <PortList>";"];
```

7.2 Declaration members

Name

This member assigns a name to the port. The name has to be unique within the application and must follow the rules for identifiers, see 2.2.

7.2.1 Task function

As the function of the task is always application dependant the function has to be implemented by the application developer. In the Zerberus code file the developer has to declare the function name followed by opening and closing brackets. The exact signature of the function is dependant on the used ports and on the Zerberus run-time system and can be stated by the code generator for a given combination of application and run-time system.

7.2.2 Port list

There are different possibilities for a task to access a port: a read-only, a write-only or a read-write access. The read access takes always place at the logical begin of a new task function call. The results are written at the logical end of the task into the ports with write access.

The port lists are therefore sorted into three different lists:

- in: The in list contains all ports that are accessed by the task only to read
- inout: The inout list contains all ports that are read and written by the task
- out: The out list contains all ports that are written by the task

The order of the lists and the order of the ports contained in each list can be chosen arbitrarily. Nevertheless the order of the lists and of the ports contained in the list is reflected in the order of the function parameters. It is forbidden that a port is contained multiple times in the port lists of one task. In addition all ports that are contained in the port lists have to be declared before the task declaration.

It is not necessary to specify all port lists, but a task needs at least one read and one write access.

7.2.3 Guard

It may sometimes be useful to execute tasks only under certain circumstances. This approach can be realized by using guards to combine the start of a task with an evaluation function based on port values. This evaluation function is realized by guards. To combine a task with a guard, the name of the guard must be specified in the task declaration. The guard has to be declared in advance. The use of the guard declaration member is optional.

7.3 Errors

7.3.1 Incomplete declaration

The code generator will reject a task declaration if one of the non-optional members is missing.

7.3.2 No references

If the task is not used in any mode the code generator will reject the Zerberus code file.

7.4 Examples

```
task task1
{
    function=taskFunction1();
    in=port1;
    out=port3;
    guard=guard1;
}

task task2
{
function=taskFunction2();
inout=port2;
out=port3;
}
```

These declarations declare two tasks task1 and task2. The first task uses the taskFunction1() for the computation. It reads the port1 and writes the results in port3. Before every start of task1 it is tested if the guard evaluates true. If the evaluation returns false, then the start is postponed until the next logical start point in time.
The second task uses the function taskFunction2() for the computation. It reads port2 and writes the result in the ports port2 and port3. No guard decides about the execution of the task.

8 Modes

Composition of tasks, actors and sensors

A mode is a set of tasks, actors and sensors that are currently executed by the system. The mode declaration specifies the according tasks, actors and sensors and assigns each object a frequency. The frequency determines how often an object is executed in one mode cycle. The duration of one mode cycle is fixed by the mode duration declaration member. The resolution of this member is nanoseconds to be in conformance with POSIX. Nevertheless the actual achievable resolution is run-time system dependant.

As the application can consist of several modes exactly one mode has to be assigned as start mode. The current mode can be changed at the end of each mode cycle, see modechanges in section 9.

8.1 Syntax of mode declaration

```
<ModeDeclaration> = mode <ModeName>
                    "{"
                       ["startmode;"]
                       ["task =" <TaskList>";"]
                      ["actor =" <ActorList>";"]
                       ["sensor =" <SensorList>";"]
                       duration = <DurationTime>";"
                    "}";

<ModeName> = <Identifier>;
<TaskList> = <TaskName> <Frequency>
                  {","<TaskName> <Frequency>};
<ActorList> = <ActorName> <Frequency>
                  {"," <ActorName> <Frequency>};
<SensorList> = <SensorName> <Frequency>
                  {"," <SensorName> <Frequency>};
```

```
<Frequency>=<Number>
<DurationTime>::=(<Number>" s," ["," <Number>" ns"]";") |
                  (<Number>" ns," ["," <Number>" s"]";");
```

8.2 Declaration members

8.2.1 Name

This member assigns a name to the mode. The name has to be unique within the application and must follow the rules for identifiers, see 2.2.

8.2.2 Start mode

Exactly one mode has to be assigned as start mode and its declaration must contain the startmode flag.

8.2.3 Task list

The task list member specifies the tasks executed in the mode and their frequency. The frequency is a positive integer value and determines the number of task executions within one mode cycle if the potential guard always evaluates true.

The task list member is optional. A possible reason to declare a mode without any task could be to wait for a certain sensor signal.

8.2.4 Actor list

The actor list allows the declaration of the actors used in the mode with the corresponding frequency. The frequency has to be a positive integer value and determines the number of actor executions within one mode cycle.

The actor list member is optional. A possible reason to declare a mode without any actors could be to compute intermediate results.

8.2.5 Sensor list

Within the sensor list the sensors that are executed within the mode are specified. A positive integer value is assigned to every sensor as frequency. This number determines the number of sensor executions within one loop cycle.

The sensor list is optional. A possible reason to declare a mode without any sensor is to perform computations based on intermediate results.

8.2.6 Mode duration

The non-optional member mode duration must be used to specify the duration of one mode cycle in seconds and nanoseconds. The achievable time resolution is run-time system dependant. For details refer to the documentation of the appropriate run-time system.

8.3 Errors

Incomplete mode declaration For a correct mode declaration at least the mode duration has to be specified.

8.3.1 Non-deterministic port values

To guarantee a deterministic execution of the mode, it is necessary to forbid that one port is written simultaneously by more than one sensor or task. The code generator performs tests to check the compliance with this condition. If a conflict is inevitable the code generator will refuse the mode declaration. But it is also possible that a conflict is not exactly predictable. This is for example the case if two tasks controlled by guards write the port. In this case a conflict can only be detected at run-time. The code generator will therefore generate a warning to signal the developer a potential error source. At run-time a violation of this rule will be intercepted and the normal system execution aborted.

8.3.2 Missing or ambiguous start mode

A possible error that leads to a rejection of the Zerberus code file by the code generator is the absence of a start mode or the existence of several start modes.

8.3.3 Missing support of duration times and frequencies

The achievable control response times and with it the mode duration time and the maximal frequency of tasks, sensors and actors are run-time system dependant. For details please refer to the documentation of the run-time system. The code generator performs tests to check the compliance of the application requirements against the run-time system limitations. In case of a discrepancy the code generation process is terminated with a corresponding failure message.

8.4 Examples

```
mode mode1
{
startmode;
task= task1 2;
actor= actor1 2;
sensor= sensor1 1;
duration= 100000000 ns;
}

mode mode2
{
task = task2 3;
actor= actor1 1;
duration= 1s, 500000000 ns;
}

mode mode3
{
task= task1 2, task2 1;
actor= actor1;
duration= 1000000 ns;
}

mode mode4
{
task=task1 1, task2 2;
actor= actor2;
duration= 1000000 ns;
}
```

These declarations declare four modes. The duration of the first mode is 100 milliseconds. The duration between internal time points arises from the mode cycle duration divided by the number of internal time points, the least common multiple of the frequencies. In the case of mode1 the duration is 50 milliseconds. This duration is crucial for the test of the compliance with the timing limitations of the run-time system. Within one mode cycle duration the task1 is executed twice (depending on the assigned guard guard1), the actor1 and the sensor1 are executed once.

Mode mode1 is established as start mode.

The second mode has a loop cycle duration of 1,5 seconds. The duration between internal time points is therefore half a second. The task2 is executed three times in a mode cycle the actor2 once.

The modes mode3 and mode4 seem to be very similar, but apart from timing limitations (we assume that the timing limitations are very loose) only one mode declaration is a valid. In both modes the tasks attempt to write the same port port3 at the second internal time point. But only one task is controlled by a guard: task1.

A possible reason for this attempt could be the following in mode 3: the computation of task2 is quite long and therefore needs more time. To limit the control response time task1 computes a faster approximation algorithm and writes it in the port p2 at the internal time point 1. The second execution of task1 is blocked by the guard.

Because of this possible reason the code generator will only generate a warning message to remind the user to verify that both tasks will not attempt to write the port at the same time.

For mode4 on the other hand the above example is not valid. Because task2 has no guard, task2 will write the port3 at both internal points. The only possible way to avoid the conflict would be to block task1 all the time. In this case the assignment of task1 to mode4 would be useless. The code generator assumes an error by the application developer and aborts the code generation with a failure.

9 Modechanges

Switching the current mode

Modechanges are used to control the mode execution. In a modechange declaration a function is assigned to the modechange to evaluate if the modechange should be executed. The function computes the decision on the base of the assigned ports. A possible modechange is evaluated always at the end of its source mode cycles. In case of a positive evaluation the next mode that is executed is the target mode of the modechange.

Currently the modechange function has to be implemented by the application developer. In the future it is envisaged to specify the function directly in the development tool.

9.1 Syntax of modechange declaration

```
<ModechangeDeclaration>  = "modechange "<ModechangeName>
                           "{"
                              "function =" <Function>";"
                              ["in =" <PortList>";"]
                              "source =" <ModeList>";"
                              "target =" <ModeName>";"
                           "}";

<ModechangeName> = <Identifier>;
<ModeList> = <ModeName> {","<ModeName>};
```

9.2 Declaration members

9.2.1 Name

This member assigns a name to the modechange. The name has to be unique within the application and must follow the rules for identifiers, see 2.2.

9.2.2 Function

The modechange evaluates if the modechange should occur. The modechange function member of the declaration determines the name of the function. The function has to be implemented by the developer. In the Zerberus code file only the function name followed by a left and a right parenthesis has to be stated since the exact signature is dependant on the ports and the run-time system. The signature can be specified by the code generator for a given combination of application and run-time system.

9.2.3 Port list

The modechange evaluation function can use port values as base of the computation. In the port list member of the modechange declaration the appropriate ports are specified. All ports used in the list have to be declared before the modechange declaration.

The port list is optional. If no ports are used as base for the function, the function should always evaluate true.

9.2.4 Source mode list

The potential modechange can be tested at the end of the mode cycle of different modes. The individual modes are specified in the source mode list. The list must contain at least one mode. Each used mode has to be declared in advance.

9.2.5 Target mode

If the evaluation function returns true the next mode executed by the system is the target mode of the modechange. The target mode is specified in the according member of the modechange declaration and has to be declared in advance.

9.3 Errors

9.3.1 Incomplete declaration

The mode change declaration is incomplete if one or more non-optional items (function, source mode list, target mode) are missing.

9.3.2 Non-deterministic mode selection

To guarantee a deterministic system execution the mode selection must be deterministic. To ensure this determinism it is forbidden that more than one modechange evaluates true simultaneously. Since the compiler can not perform a semantic check of the modechange function implemented by the developer, this condition must be checked by the developer. The compiler will generate a warning if there is a possibility for this conflict to point up the hazard.

9.3.3 Non-reachability of a mode

Another error source related to modechanges is the non-reachability of a mode. This can happen if a mode is not assigned as startmode and no modechange contains the specific mode as target mode. Since a non-reachable mode is useless an error is assumed. The compiler performs a test to check the reachability of each mode. The according algorithm uses the assumption that there exists a port value configuration for every modechange so that the modechange evaluates true. This assumption is reasonable since otherwise the modechange would be useless. On the other hand the code generator can not perform the semantic check of the modechange.

9.3.4 Invalid port compare state

Every port that is read by a modechange must be compared during the voting algorithm execution. If this condition is not fulfilled the Zerberus code generator will state an error and stop its execution.

9.4 Examples

```
modechange modechange1
{
function= shouldChangeMode1();
in=port3;
source=mode1,mode2;
target=mode3;
}

modechange modechange2
{
function= shouldChangeMode2();
source=mode3;
target=mode2;
}
```

In this example two modechanges are declared. The first modechange modechange1 changes the modes from mode1 or mode2 if the evaluation function shouldChangeMode1 evaluates true with the function parameter port3.
The second mode change switches the mode from mode3 to mode2. Since no port is stated the evaluation function always returns the same value and in order to be not useless should evaluate true every time.
Because mode1 is declared as start mode, every mode should be reachable under the assumption of correct and reasonable evaluation functions.

10 Functions

Functions implemented by the application developer

Several functions have to be implemented by the application developer:

1. Compare functions
2. Initialization functions
3. Actor functions
4. Sensor functions
5. Guard functions
6. Task functions
7. Modechange functions

The functions have to be implemented in the same programming language as the run-time system. For future versions of the Zerberus system it is envisaged to specify all the functions except the actor, sensor and task functions within the development environment. Actor and sensor functions must of course only be implemented if the according sensor or actor is not supported by the run-time system.

The following subsections describe the different function types, the functionality and their parameters. The exact signature and the return parameter is platform dependant and can be specified by the code generator for a given combination of application and run-time system. For more detailed information please refer to the documentation of the run-time system.

The reuse of functions in only allowed for the following cases: functions for initialization and comparison of ports, guard and modechange functions as well as sensor and actor functions. The reuse of task functions is forbidden. If different objects except tasks use a function with the same name, the type of the function and the signature (depending on the used ports) must be identical.

10.1 Compare function

If a bit by bit comparison of the port value is not reasonable the application developer can implement an own compare function. These functions do often realize an interval check. If the difference of the port values is below a predetermined threshold the comparison is successful, otherwise an error is indicated. An interval check can be useful if the computation is not deterministic or if the input values can differ because of a slight temporal difference of the units.

The execution of the compare function is time-critical and should logically consume zero time. Therefore long computation times are forbidden.

The parameters of the function are the two values of the same port on different units.

10.2 Initialization function

To enable a correct voting every port has to be initialized. The initial value can be stated directly in the Zerberus code file. Another possibility is to use initialization functions. The use of these functions can be useful if more ports of the same type have to be initialized in the same way or to guarantee the compactness of the Zerberus code file.

The parameter of the non-time-critical initialization functions is the related port.

10.3 Actor function

Actors and their functions realize the system output to the environment on the base of port values. Several devices are supported by the Zerberus system but to guarantee a general usability the Zerberus system allows the implementation of own actor functions if the desired actor is not available.

The parameter of the actor functions are specified in the port list of the according actor. Since the execution of the actor function is time-critical, it should logically consume zero time. The actor function should therefore execute only short copy and transformation operations. Especially a blocking of the actor function is not allowed.

10.4 Sensor function

Sensors and their functions realize the system input from the environment into the ports. Several devices are supported by the Zerberus system but to guarantee a general usability the Zerberus system allows the implementation of own sensor functions if the desired device is not available.

The parameters of the sensor functions are the ports as specified in the port list of the according sensor. Since the execution of the sensor function is time-critical, it should logically consume zero time. The sensor function should therefore execute only short copy and transformation operations. Especially a blocking of the sensor function is not allowed.

10.5 Guard function

Guards and their functions control the task execution. The decision if a task is executed is based on the values of the specified ports. The ports are therefore the parameters of the function.

Since the execution of the guard function is time-critical, it should consume logically zero time. No guard function should therefore perform long computations.

10.6 Task function

Task functions represent the real application functionality. Based on the input ports the results are computed and written into the output ports. The parameters of the task function are therefore the ports as specified in the port lists of the task.

The task function execution is time-critical and has to meet the timing restrictions as specified by the application developer. A blocking of the task function is not allowed. This presents no restriction to the task function since it should be a pure computation function without any synchronization points.

10.7 Modechange function

By using modechanges the current mode that is executed by the system can be selected. The decision is reached by the evaluation of the modechange function based on the values of the ports as specified in the port lists of the according modechange. These ports are therefore the parameters of the modechange function.

The execution of the modechange function is time-critical and should logically consume zero time. It is therefore forbidden to perform long lasting computations or to block the function.

A Basic syntactical definitions

Basic definitions used in preceding sections

A.1 Identifier

The conventions for identifier extracted from general programming languages. If there are some ambiguities with other programming languages the identifier has to be chosen this way that the identifier is supported both by the Zerberus language as by the programming languages used by the employed run-time systems. The identifier must be unique within the application and it is forbidden to use keywords of the Zerberus language (see section B) and of the programming languages used to implement the deployed run-time systems.

```
<Identifier> = <Letter> {<Letter>|<number>|"_"};
<Letter> = ’A’|’B’|’C’|’D’|’E’|’F’|’G’|’H’|’I’|’J’|’K’
   |’L’|’M’|’N’|’O’|’P’|’Q’|’R’|’S’|’T’|’U’|’V’
   |’W’|’X’|’Y’|’Z’|’a’|’b’|’c’|’d’|’e’|’f’|’g’
   |’h’|’i’|’j’|’k’|’l’|’m’|’n’|’o’|’p’|’q’|’r’
   |’s’|’t’|’u’|’v’|’w’|’x’|’y’|’z’;
<Number> = <Digit> {<Digit>};
<Digit>= ’0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’;
```

A.2 Function

At certain positions in the Zerberus code the application developer has to refer to a run-time dependant function. Since the Zerberus language is independent from a certain programming language, the exact signature is not previously known. Because of this reason the application developer must only specify the function name followed by a left and a right parenthesis.

```
<Function> = <Identifier>"()"
```

B Keywords

Reserved words

The following words are reserved and are not valid as identifier:

port	actor	sensor	guard	task	modechange	mode
in	out	inout	function	duration	s	ns
startmode	source	target	arraySize	initialValue	compareMode	binary
none	type	BOOL	CHAR	UCHAR	INT16	INT32
INT64	UNIT16	UINT32	UINT64	FLOAT32	FLOAT64	