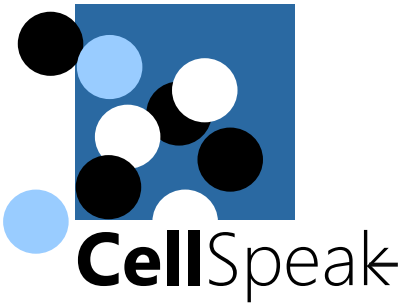

CellSpeak

Platform Package

July 2017

Cells run in parallel,
On all cores,
On interconnected systems,
As one program

Table of Contents

1 Introduction

1.1 Packages and modules

A project in CellSpeak compiles to a module, and modules can be combined together in packages, i.e. a package can contain one or several modules. It is practical to combine related modules into a single package, or even to put a complete application in a single package, but the 'unit of dependency' in a CellSpeak program, is the module. Sometimes the terms *package* and *module* are used interchangeably, where in most cases the term *module* should have been used. Most of the time this does not lead to confusion.

Note that modules are identified by their *group* name in the source code, but not every group is a module – a module can indeed contain many groups.

1.2 Strong and weak links

This manual describes the packages that are available for development in CellSpeak.

Packages contain modules and modules contain types, functions and designs that are useful for many other projects.

Modules that share types with other modules are said to have a strong link, meaning that if the types in these modules change, then also the modules that use these types have to be recompiled.

Modules that only expose designs and interfaces to be used by other modules for instantiation and exchange of messages, are said to have weak links. When these packages change, the packages that use them do not have to be recompiled.

Of course, if the interface of a design – the messages it can receive and send – changes, then the module that wants make use of these changes will have to be modified and recompiled. Additions or changes to the internal workings of weakly linked modules, have no effect on the modules that use these modules.

It is clear that it is preferable to use in a project as much as possible weakly linked modules. Some modules however mostly contain types and functions, such as the *math* or *string* library. These packages are ‘standard’ packages and not expected to change often so they can be safely used by other packages without worrying too much about the dependency.

Often however packages will need to export some type definitions and design definitions in order to make their full functionality available. In that case it can be beneficial to separate the module in two or more modules: one module for the type definitions and one or more modules for the designs and interfaces. All these modules can of course conveniently be combined into one package.

1.3 Format of the description

In the following chapters we describe the messages that can be received by the interfaces supported by the different cell types and the replies that these cells can give to these messages:

```
design NameOfTheDesign( parameters ) is

          interface NameOfTheInterface is

             on MessageA( parameters ) do

                   <- ReplyA1( parameters )
                   <- ReplyA2( parameters )

             on MessageB( parameters ) do

                   <- ReplyB1( parameters )
                   <- ReplyB2( parameters )

             etc.
```

The parameters are given as a combination of type and a name for the parameter. As we have seen, the name of the parameter has no significance in the selection of a message handler and is added here just for clarity. The developer can choose whatever name he wants for a message parameter.

1.4 Overview

The *Platform Package* contains types, functions and designs that are common to most projects. It offers an interface into the services provided by the OS and to the services provided by the Virtual Machine.

When a Virtual Machine is launched, it will create a cell *system*, which is of the type *SystemDesign.* Note that *system* is a reserved keyword in CellSpeak. The cell *system* offers a number of interfaces and also implements an interface to offer and to obtain additional services.

Also part of the platform package are the groups *math* and *strings* and *structures.* The *structures* group contains templates for commonly used containers, like a linked list or a hash table, that can be instantiated and extended.

The Platform Package consists of the following designs and libraries

- The system
- The connection manager
- Timer service
- The math library
- The strings library
- The structures library
- File handling library
- Windows (OS specific services and designs)

In the next paragraphs we take a closer look at the components of the platform package.

A service is always offered by a cell or a group of cells, whereas a library is a collection of types and functions that can be re-used by the developer.

2 The system

2.1 Overview

The following is an overview of the *system* and the interfaces it supports. For each message also the possible return messages are given. Note that the *system* cell is created automatically by the Virtual Machine and does not need to be created by the user.

```
design SystemDesign is

          interface Service is

             on Get(utf8 Name) do

                    <- Service.Provider(utf8 Name, cell Provider)
                    <- Service.NoProvider(utf8 Name )

             on Offer(utf8 Name, cell Provider) do

                    <- Service.InvalidProvider(utf8 Name )
                    <- Service.Exists(utf8 Name, cell Provider)
                    <- Service.Added(utf8 Name, cell Provider)
                    <- Service.NotAdded(utf8 Name, cell Provider)

             on Revoke(utf8 ServiceName, cell ServiceProvider)

          interface end

          interface Stdout

             on Set(cell Stdout)
             on Get(cell Stdout)

                    <- Stdout(cell Stdout)

             on Print(utf8 Line)
             on Println(utf8 Line)

          interface end

          interface Stderr

             on Set(cell Stderr)
             on Get(cell Stderr)

                    <- Stderr(cell Stderr)

             on Print(utf8 Line)
             on Println(utf8 Line)

          interface end

          on Print(utf8 Line)
          on Println(utf8 Line)
          on Error(utf8 Line)
```

```
          on Memory.GetStatus(cell Cell)

             <- Memory.Usage(CellMemoryUsage Usage)

          on ByteCode.Load( byte[] ByteCode)

             <- ByteCode.Load.Ok
             <- ByteCode.Load.Failed

          on Creator.Get

             <- Creator( cell Creator )


end
```

2.2 The Service interface

The service interface allows to offer and to obtain a service from the system. The following gives an overview of the messages the interface supports and the replies that can be given.

```
interface Service is

  on Get(utf8 Name) do

        <- Service.Provider(utf8 Name, cell Provider)
        <- Service.NoProvider(utf8 Name )

  on Offer(utf8 Name, cell Provider) do

        <- Service.InvalidProvider(utf8 Name )
        <- Service.Exists(utf8 Name, cell Provider)
        <- Service.Added(utf8 Name, cell Provider)
        <- Service.NotAdded(utf8 Name, cell Provider)

  on Revoke(utf8 ServiceName, cell ServiceProvider)

interface end
```

A service can be obtained from the system using the *Get* message with the name of the service as its only parameter. Example

```
system <- Service.Get("TimerService")
```

If the service is available, the system will reply by sending the message *Service.Provider*. The parameters are the again the name of the service and the cell that provides the service. The name of the service is repeated in the reply so that, in case the developer has requested several services, he can make the link to the service requested.

If a service is not available, the system replies with *Service.NoProvider*, with the name of the requested service as the only parameter.

A service is offered to the system using the *Offer* message.

2.2.1 Service.Get→

Service.Get(utf8 Name)	
description	Request the system for a service of the given name. The system will reply by sending the cell that provides the service, if available
parameters	• utf8 Name: the name of the service
replies	← Service.Provider(utf8 Name, cell Provider) ← Service.NoProvider(utf8 Name)
remarks	-

2.2.2 Service.Offer →

Service.Offer(utf8 Name, cell Provider)	
description	Makes a service available to other cells.
parameters	• utf8 Name: the name of the service • cell Provider: the cell that provides the service
replies	← Service.InvalidProvider(utf8 Name) ← Service.Exists(utf8 Name, cell Provider) ← Service.Added(utf8 Name, cell Provider) ← Service.NotAdded(utf8 Name, cell Provider)
remarks	A cell can offer a service that is provided by another cell, typically a child cell.

2.2.3 Service.Revoke →

Service.Revoke(utf8 Name, cell Provider)	
description	Revokes a service that was previously made available.
parameters	• utf8 Name: the name of the service • cell Provider: the cell that provides the service
replies	
remarks	The cell that revokes the service must be the same that has offered the service.

2.2.4 ← Service.Provider

Service.Provider(utf8 Name, cell Provider)	
description	Returns the cell that provides a service to the cell that requested the service.
parameters	• utf8 Name: the name of the service that was requested • cell Provider: the cell that provides the service
remarks	

2.2.5 ← Service.NoProvider

Service.NoProvider(utf8)	
description	Reply if there is no provider for the requested service

parameters	• utf8 Name: the name of the requested service
remarks	

2.2.6 ← Service.InvalidProvider

	Service.InvalidProvider(utf8 Name)
description	Reply if the cell that offers the service is not valid.
parameters	• utf8 Name: the name of the requested service
remarks	

2.2.7 ← Service.Exists

	Service.Exists(utf8 Name, cell Provider)
description	Reply if the service offered by a cell is already available
parameters	• utf8 Name: the name of the requested service • cell Provider: the cell that provides the service
remarks	

2.2.8 ← Service.Added

	Service.Added(utf8 Name, cell Provider)
description	Reply if the service was successfully added to the list of services
parameters	• utf8 Name: the name of the requested service • cell Provider: the cell that provides the service
remarks	

2.2.9 ← Service.NotAdded

	Service.NotAdded(utf8, cell Provider)
description	Reply if the service was not added to the list of services
parameters	• utf8 Name: the name of the requested service • cell Provider: the cell that provides the service
remarks	

2.3 The Stdout and Stderr interfaces

The Stdout and Stderr interfaces are used to provide a simple way of outputting messages to the user.

```
interface Stdout

  on Set(cell Stdout)
  on Get(cell Stdout)

        <- Stdout(Stdout)

  on Print(utf8 Line)
  on Println(utf8 Line)

interface end

interface Stderr

  on Set(cell Stderr)
  on Get(cell Stderr)

        <- Stderr(Stderr)

  on Print(utf8 Line)
  on Println(utf8 Line)

interface end

on Print(utf8 Line)
on Println(utf8 Line)
on Error(utf8 Line)
```

The user has to provide a cell that implements two simple messages: Print and Println. The second message prints the output on a new line. There are actually three ways to print a line to the standard output:

1. Get the Stdout cell from the system and send the Print messages directly to that cell:

```
Stdout <- Print( … )
```

2. Send a Print message to the Stdout interface of the system cell:

```
system <- Stdout.Print( … )
```

3. Send the Print message directly to the system

```
system <- Print( … )
```

In the last two cases the system will relay the message to the Stdout cell that has been set by the user.

If Stdout or Stderr have not been set, the messages are silently discarded.

2.3.1 Stdout.Set →

description	
parameters	
replies	
remarks	

2.3.2 Stdout.Get →

description	
parameters	
replies	
remarks	

2.3.3 ← Stdout

description	
parameters	
replies	
remarks	

2.3.4 Stdout.Print →

description	
parameters	
replies	
remarks	

2.3.5 Stdout.Println →

description	
parameters	
replies	
remarks	

2.3.6 Stderr.Set →

description	
parameters	
replies	
remarks	

2.3.7 Stderr.Get →

description	
parameters	
replies	
remarks	

2.3.8 ← Stderr

description	
parameters	
replies	
remarks	

2.3.9 Stderr.Print →

description	
parameters	
replies	
remarks	

2.3.10 Stderr.Println →

description	
parameters	
replies	
remarks	

2.3.11 Print →

description	
parameters	
replies	
remarks	

2.3.12 Println →

description	
parameters	
replies	
remarks	

2.4 Memory interface

The memory interface has one message

```
          on GetStatus( cell Cell )
                   <- Memory.Usage( Usage )
```

The system replies with the Memory.Usage message with one parameter, a record of the type *CellMemoryUsage:*

```
type CellMemoryUsage is record
          int      NrOfItems
          int      BytesAllocated
end
```

2.4.1 ByteCode interface

The ByteCode interface supports one message:

```
          on ByteCode.Load( byte[] ByteCode) do

             <- ByteCode.Load.Ok
             <- ByteCode.Load.Failed
```

The ByteCode parameter is an array of bytes that contains the bytecode that has to be loaded by the system. Typically the bytecode in the array will have been loaded from a file, or transmitted to the system via the network. If the ByteCode was loaded successfully, the system replies with the *ByteCode.Load.Ok* message, if not the system sends *ByteCode.Load.Failed*.

2.5 Creator interface

Cells can be created in two ways on the system: either by using the keyword *create*, in which case the resulting cell is created as a child of the cell that created it, or by sending a creation request to the *Creator* cell.

Cells that are created by the Creator cell are added to the root of the cell-tree. This way of creating cells is especially useful to instantiate an application on a system. We will see that it is also possible to get a reference to the creator cell of a remote system, which allows then to instantiate cells (applications) on that remote system. Combined with the Bytecode interface, this is a way to load and execute bytecode on a remote system.

The Creator cell can be obtained from the system using the *Creator.Get* message

```
on Creator.Get

   <- Creator( cell Creator )
```

The system replies with the *Creator* message which has one parameter, the cell that can create other top-level cells (applications).

The way to create a cell using the *Creator* cell is the same as using the create keyword, but the design and its parameters are sent as a message to the *Creator:*

```
Creator <- abc( p1, p2, p3 )
```

The creator cell will then create a cell of the design abc using the parameters p1 ,p2 and p3 and add that cell to the root of the cell-tree of the virtual machine.

Note that

```
create abc(p1, p2, p3)
```

would create exactly the same cell, but this time as a child cell of the cell who created it.

3 The Connection Manager

The Connection Manager – or ConxMgr in short – handles incoming and outgoing connections.

It can be asked to connect to other systems or to listen for incoming connections using several communications protocols. The ConxMgr will take care of establishing the link and the buffering required when sending and receiving data. It will inform its clients of events, e.g. when a link gets disconnected.

The connection manager can use several protocols. It can communicate between remote systems running CellSpeak virtual machines and exchange messages between these systems in the same format as if the cells on the remote machine were local.

The ConxMgr can also exchange messages in HTTP format, for example to communicate with the browser or to communicate with a system using a particular RESTful API of the system.

The ConxMgr can also encrypt all communications using SSL.

As we will see, the ConxMgr is easy to use, much easier than in traditional programming languages, because the message oriented nature of CellSpeak is a natural fit for handling communications: asynchronous events, buffering, parallel activities – are part and parcel of the message switching architecture of the CellSpeak Virtual Machine.

As can be expected, the ConxMgr is a cell. There is one ConxMgr in a running Virtual Machine and it is created by the *system* cell when the Virtual Machine is started. The ConxMgr can be obtained from the system as a service with the name *ConxMgr* as follows

```
system <- Service.Get("ConxMgr")
```

The system will respond then as follows:

```
sender <- Service.Provider(“ConxMgr”, Provider)
```

where *Provider* is the cell-id of the ConxMgr.

In the following paragraphs, we will take a look at how two systems can setup a communication over TCP/IP between two systems running CellSpeak, how to setup a communication link with a browser and how to handle the communication with a site using a published API.

In the following discussions, it is assumed that the reader has a basic understanding of the protocols used (TCP/IP, SSL, HTTP).

Note that the ConxMgr is a cell but also a collection of objects and methods that take care of communications that are executed by the avatars (more about avatars later). In that respect, the ConxMgr is as distributed and parallel as the any other CellSpeak application. Communications between systems take place in parallel and the ConxMgr is not a bottleneck.

3.1 Communicating between CellSpeak virtual machines

Setting up a connection between two systems only requires a few messages:

```
Caller                          ConxMgr / ConxMgr              Callee


Connect to ‘system:port’   ------->          <-------     Listen on ‘port’
Connected                  <-------          ------->     Accept conx on ‘port’
SetProtocol                ------->          <-------     SetProtocol
Protocol.Ready(callee)     <-------          ------->     Protocol.Ready(caller)
```

The caller sends a connection request to the ConxMgr, specifying the name, or IP address, of the remote system and the port number it wants to connect to.

On the remote system, the cell that is willing to accept incoming connections, informs its ConxMgr about the port number it will listen to.

Both ConxMgrs, the one at the caller’s site and the one at the callee’s site, will then set up the communication between both systems. When the connection has been established, the cell that has initiated the connection gets a *Connected* message and the cell listening to the port number will get an *Accept* message.

Both sides of the communication then have to specify the user-level protocol they want to use over the connection. In this example both sides would specify the CLSPK – CellSpeak – protocol. The connection manager would then confirm the selection to both by sending a *Protocol.Ready* message with the id of the remote cell as a parameter. From then onwards the cells can start communicating between themselves directly in exactly the same way as if they were local cells.

The two cells at either side of the comms channel can for example exchange other cell-id’s to communicate with, and in this way many cells on one side of the channel can communicate with many cells at the other side. As a side remark: it is one of the tasks of the communication manager to translate remote cell-id’s to local cell-id’s to avoid collisions between cell-ids’. The local cell-id for a remote cell is often referred to as the *avatar* of that cell.

If anything happens to a communication link, for example if it gets disconnected, the ConxMgr will inform the two cells that have set up the communication about the event.

It is often convenient to split the responsibility for the communication set-up and the actual message-exchange over a connection, over different cells. This can easily be done, because the *SetProtocol* message allows to specify the cell that will handle the message exchange between the two systems. In this way many calls can be setup using the same port number, each time establishing a direct communication between different cells at either side of the link.

3.2 Communicating using the HTTP protocol

Setting up a communications link between two systems to use the HTTP protocol follows the same steps as above, apart from the last step where the *SetProtocol* message will specify the HTTP protocol iso the CLSPK protocol.

The cell can then send HTTP messages using the familiar CellSpeak format, e.g.

```
avatar <- HTTP.RESPONSE("HTTP/1.1", "200", "OK", ResponseHeaders, StyleSheet)
```

The ConxMgr will translate this CellSpeak format into the appropriate HTTP format before transmitting it to the remote system, represented on the local system by its *avatar.*

Conversely, incoming HTTP messages from the remote system will be formatted transparently into the familiar CellSpeak format, e.g.:

```
on GET(ansi URI, ansi Version, Header[] Headers, byte[] Content) do
            …
end
```

where the different components of the HTPP message are split into a message name (GET) and several other data structures.

The use of SSL – Securce Sockets Layer – is completely transparent to the CellSpeak program – it has only to be mentioned once at the setup of the communication. The ConxMgr will take care of the encryption/decryption of the communication between the two systems.

3.3 Overview

```
design ConxManagerDesign( cell owner ) is

          interface TCPIP is

             on Connect(byte[] Host, byte[] Port)

                    <- Connected( ConxClass Conx )

             on Close(cell Avatar)
             on Listen(byte[] Port)

                    <- Accept( ConxClass Conx )

             on Stop.Listening(byte[] Port)
             on SSL.Connect(byte[] Host, byte[] Port)
             on SSL.Listen(byte[] Port)

             -- message sent when disconnected by the remote party

                    <-TCPIP.RemoteDisconnected( ConxClass Conx )

          interface end

          on Set.Protocol( ConxClass Conx, ansi Protocol, cell Handler)

                    <- "[Protocol].Ready"(cell Avatar)
                    <- "[Protocol].NotSupported"


          on Change.Handler( ConxClass Conx, cell NewHandler)

                     <- Change.Handler.Ok
                     <- Change.Handler.Failed
```

The *ConxClass* is an opaque reference to the connection that is used by the ConxMgr to identify the link.

3.4 HTTP Messages

In order to handle HTTP messages in a CellSpeak application, the ConxMgr formats incoming HTTP messages into a CellSpeak format and formats outgoing CellSpeak messages into an HTTP format.

3.4.1 Incoming HTTP messages

When an HTTP message is received, the corresponding CellSpeak message gets the name of the HTTP message. The content of the HTTP message is packed into four parameters:

```
interface HTTP is

          on GET(ansi URI, ansi Version, Header[] Headers, byte[] Content)
```

- URI : The universal resource identifier
- Version : the version of the HTTP protocol in string format, e.g. "HTTP/1.1"
- Headers : an array of *name, value* pairs, both as strings. Example:

```
ResponseHeaders = [ [ "Date", HTTPDate()],
                    [ "Content-Type", "text/html" ],
                    [ "Content-Length", "[MyMessage.len()]" ]  ]
```

- Content : a byte array that carries the payload of the message

3.4.2 Outgoing HTTP messages

Outgoing HTTP messages are constructed in the same way as incoming messages. The name of the message is the HTTP name of the message with the headers and content added as parameters. Example:

```
ResponseHeaders = [ [ "Date", HTTPDate()],
                    [ "Content-Type", "text/html" ],
                    [ "Content-Length", "[nel HomePage]" ]  ]


sender <- HTTP.RESPONSE("HTTP/1.1", "200", "OK", ResponseHeaders, HomePage)
```

The message above is a response message. The first three parameters are the version, the HTTP code indicating success or failure and a textual message corresponding to the code. The next parameters are the headers as *name, value* pairs and finally the payload of the message, in this case called *Homepage*, and presumably a byte array of HTML code.

3.5 Using SSL

The use of the *Secure Socket Layer* only makes a difference when the connection between two systems has to be established: instead of *Connect* and *Listen*, the messages *SSL.Connect* and *SSL.Listen* are used. Once the connection is established, exchange of messages is as in the non-secure case.

CellSpeak uses the well-known OpenSSL toolkit for the implementation of the secure communication.

4 Timer

The timer is a service made available by the Virtual Machine at startup. It has a simple and intuitive interface.

The timer service can be obtained from the system using the *Service* interface:

```
system <- Service.Get("Timer")
```

The system will then respond with

```
<- Service.Set("Timer", TimerCell)
```

where *TimerCell* is the cell that implements the timer functions.

The timer has two interfaces, *Subscribe* and *Unsubscribe*, as shown below:

```
design TimerDesign is

   interface Subscribe is

      on Interval( word Interval, utf8 ReturnMessage ) do
      on Relative( word TimeDelta , utf8 ReturnMessage )  do
      on Absolute( word Time, utf8 ReturnMessage ) do

   interface end

   interface Unsubscribe is

      on Interval( word Interval, utf8 ReturnMessage ) do
      on Relative( word TimeDelta , utf8 ReturnMessage )  do
      on Absolute( word Time, utf8 ReturnMessage ) do

   interface end
```

A cell can request for three sorts of timer events : *Interval, Relative* and *Absolute*. Each request has two parameters: the time in milliseconds and the name of the message to receive when the timer expires.

A *Relative* and an *Absolute* timer will fire only once, whereas an interval timer will fire after each interval.

In the following example the timer is requested to send a message *FrameTick* every 40 ms:

```
TimerCell <- Subscribe.Interval( 40w, "FrameTick" )
```

The cell will have provide a handler for the message from the timer with the following signature

```
on FrameTick( word Time ) do …
```

The parameter *Time* is the time in msec since the Timer started.

5 Math Library

The math library provides functions, operators, constants and datatypes for math applications. There is no service or cell associated with math, so the math library is a pure library.

The functions and types in the math library are grouped as follows:

- Real valued functions
- Integer functions
- Complex numbers
- Vector functions for vectors of 2, 3 or 4 components
- Matrix functions for 2x2, 3x3 and 4x4 matrices
- Rotations
- Quaternion functions

The math library is written for performance. Most of the simple functions, like the sine, cosine or exponential function, will result in the in-lining of a single CellSpeak bytecode instruction and that instruction will – where available - be compiled to the native instruction of the processor the program is run on.

5.1 Numerical format in CellSpeak

CellSpeak supports integers, unsigned integers or words of several lengths, floats and doubles. When using a constant in a program it is sometimes clear what the type of that constant is, but sometimes it is not clear.

For example in

```
float x = 1
```

it is clear that the real number 1 is intended here. But for example in a function call:

```
sin(2)
```

it is not clear whether the parameter should be interpreted as a float or as a double value.

There are two ways to solve this. The first way is to explicitly cast the value to the desired type:

```
sin( <double> 2 )
```

This is a perfectly acceptable, but sometimes a bit heavy on notation. Therefore CellSpeak allows to use 'hints' in the numerical value themselves to determine the type of the constant. The following table gives an overview of these hints:

CONSTANTS	
decimal integer	7
	1_256_789_365 : Underscores can be used to improve readability
word	556w : w signals that the constant is a word – unsigned int.
hex integer	0xff00a9 or 0xFF00A9 : Starts with 0x and can contain a-f and A-F
hex byte	0xff : like a hex integer but with only two positions
hex byte as char	0x'a' = 0x61 , 0x':' = 0x3A etc.
float	3.1415
	3.1415568957e
	6.626069e–34 : exponentiel notation
hex float	0x.7f45ad84 the 0x is followed by a decimal point and 8 hex characters
double	3.1415926535897932d
	6.626069d–34 : exponentiel notation
hex double	0x.01457a4f55de23a6 the 0x is followed by a decimal point and 16 hex characters

The example above can then be written as follows:

```
sin(2d)
```

5.2 Real valued functions

Real valued functions operate on parameters of the type float or double. The following is an overview of the real valued functions in the math library:

```
group Math

-- some numerical constants

const double pi     = <double> 3.141592653589793238  -- Circle
const float  pi_f   = <float> pi
const double e      = <double> 2.718281828459045235  -- Euler's constant
const float  e_f    = <float> e
const double phi    = <double> 1.618033988749894848  -- Golden ratio
const float  phi_f  = <float> phi
const float  inf_f  = 0x.7f800000                    -- float infinity (ieee 754)
const double inf_d  = 0x.7ff8000000000000            -- double infinity (ieee 754)

-- Trigonometry for float

function sin(float angle) out float
function cos(float angle) out float
function tan(float angle) out float
function asin(float sine) out float
function acos(float cosine) out float
function atan(float sine, float cosine) out float

-- Exponential, log etc for float

function exp(float x) out float
function log(float x) out float
function abs(float x) out float
function sqrt(float x) out float
function randf out float                        -- random float between 0 and 1

-- Trigonometry for double

function sin(double angle) out double
function cos(double angle) out double
function tan(double angle) out double
function asin(double sine) out double
function acos(double cosine) out double
function atan(double sine, double cosine) out double

-- Exponential, log etc for double

function exp(double x) out double
function log(double x) out double
function abs(double x) out double
function sqrt(double x) out double
function randd out double                       -- random double between 0 and 1
```

5.2.1 sin cos tan

	sin(float angle) cos(float angle) tan (float angle)
description	calculates the sine, cosine or tangent of an angle given in radians.
input	• float angle: the angle in radians for which the function has to be calculated.
output	float: the value of the function
remarks	-
example	

The same functions also exist for a double precision argument. In that case also the result is a double precision value:

	sin(double angle) cos(double angle) tan (double angle)
description	calculates the sine, cosine or tangent of an angle given in radians.
input	• double angle: the angle in radians for which the function has to be calculated.
output	double: the value of the function
remarks	-
example	

5.2.2 asin acos atan

	asin(float sine) acos(float cosine) atan (float sine, float cosine)
description	calculates the angle that corresponds with the values of the sine, cosine or both.
input	• float sine: a value between -1 and +1, included. • float cosine: a value between -1 and +1, included.
output	float: the angle in radians that corresponds with the values of the sine and cosine.
remarks	The angle is between 0 and π for acos and between π/2 and -π/2 for asin.
example	

The same functions also exist for a double precision argument. In that case also the result is a double precision value:

	asin(double sine) acos(double cosine) atan (double sine, double cosine)
description	
input	• float sine: a value between -1 and +1, included. • float cosine: a value between -1 and +1, included.
output	double: the angle in radians that corresponds with the values of the sine and cosine.
remarks	The angle is between 0 and π for acos and between π/2 and -π/2 for asin.
example	

5.2.3 exp

description	
input	•
output	
remarks	
example	

5.2.4 log

description	
input	•
output	
remarks	
example	

5.2.5 abs

description	

input	•
output	
remarks	
example	

5.2.6 sqrt

description	
input	•
output	
remarks	
example	

5.2.7 randf randd

description	
input	•
output	
remarks	
example	

5.3 Integer functions

```
function inc(int i) out int is
function dec(int i) out int is
function randi out int is              -- random int between -2^31 and 2^31-1
function inc(word w) out word is
function dec(word w) out word is
```

5.4 Complex numbers

Complex numbers in the math library are derived from the built-in CellSpeak vector type vec2f:

```
-- the two field names are renamed to the familiar r(eal) and i(maginary)
type complex_f is vec2f with
          rename c1 to r
          rename c2 to i
end

-- complex is a synonym for complex_f
type complex is complex_f
```

Complex numbers can be added and subtracted like vectors which are built-in operations, but the multiplication operator for complex numbers is redefined to have the expected effect for complex numbers:

```
complex a, b

-- complex multiplaction: c.r = a.r*b.r – a.i*b.i, c.i = a.i*b.r + a.r*b.i

c = a*b
```

The corresponding type for complex numbers with double precision is also defined, this time derived from the built-in vec2d type:

```
-- the two field names are renamed to the familiar r(eal) and i(maginary)
type complex_d is vec2d with
          rename c1 to r
          rename c2 to i
end
```

as is the complex multiplication.

5.5 Vector operators and functions

Vector types with 2, 3 or 4 components of type float or double, are built-in types in CellSpeak. These built-in types are used as the basis for the definition of vector types (and also for quaternions as we will see later).

The type names chosen for vectors are xy, xyz and xyzw. For vectors of the type double _d has to be added.

For every vector type a few methods are defined: norm, normalize and length. The norm returns the normalized vector, normalize actually normalizes the vector and length returns a scalar, the length of the vector.

For the type xyz and xyz_d, the functions norm and length are also available as stand-alone functions, i.e. not as methods.

```
-. xy .-

type xy_f is vec2f with

          rename c1 to x
          rename c2 to y

          function length out float
          function norm out xy_f
          function normalize

end

type xy is xy_f                         -- define xy as synonym for xy_f

type xy_d is vec2d with

          rename c1 to x
          rename c2 to y

          function length out double
          function norm out xy_d
          function normalize

end

-. xyz .-

type xyz_f is vec3f with

          rename c1 to x
          rename c2 to y
          rename c3 to z

          function length out float
          function norm out xyz_f
          function normalize
```

```
end

type xyz is xyz_f                       -- define xyz as synonym for xyz_f

function norm(xyz v) out xyz
function length(xyz v) out float

type xyz_d is  vec3d with

          rename c1 to x
          rename c2 to y
          rename c3 to z

          function length out double
          function norm out xyz_d
          function normalize

end

function norm(xyz_d v) out xyz_d
function length(xyz_d v) out double

-. xyzw .-

type xyzw_f is  vec4f with

          rename c1 to x
          rename c2 to y
          rename c3 to z
          rename c4 to w

          function length out float
          function norm out xyzw_f
          function normalize
end

type xyzw is xyzw_f                     -- define xyzw as synonym for xyzw_f

type xyzw_d is vec4d with

          rename c1 to x
          rename c2 to y
          rename c3 to z
          rename c4 to w

          function length out double
          function norm out xyzw_d
          function normalize

end
```

As these vector types are derived from the built-in CellSpeak vector types, the standard operators for vectors can also be used on these types: addition, subtraction, scalar multiplication and division,

dot multiplication and cross multiplication. Also the multiplication with matrices are inherited from the built-in types.

5.6 Matrices

The matrices in the math library are derived from the built-in matrix types of CellSpeak. The built-in matrix types are square matrices of sizes 2x2, 3x3 and 4x4 with elements of type float and double. The components of a matrix can be accessed as c11 to c44.

Matrices can be added, subtracted, multiplied by a scalar and multiplied by another square matrix of the same rank.

The following lists the additional definitions in the math lib for the 3x3 matrices, both for matrices of floats and for matrices of doubles. The same definitions are also available for 2x2 and 4x4 matrices.

First the name of the type is redefined to *matrix3* iso of *mtx3f* with three methods added. The *transpose* and *inverse* methods are also defined as functions that do not alter the matrix parameter. Finally the identity matrix for both types are given as constants.

```
-. matrix 3x3 .-

type matrix3_f is mtx3f with

          function det out float
          function inverse
          function transpose
end

function inverse( matrix3_f m) out matrix3_f
function transpose( matrix3_f m) out matrix3_f is

type matrix3 is matrix3_f

type matrix3_d is mtx3d with

          function det out double
          function inverse out mtx3d
          function transpose
end

function inverse( matrix3_d m) out matrix3_f
function transpose( matrix3_d m) out matrix3_d

const matrix3_f I3X3 = [  1,0,0,
                          0,1,0,
                          0,0,1  ]

const matrix3_d I3X3_d = [1,0,0,
                          0,1,0,
                          0,0,1  ]
```

For the other matrix types, the 3x3 types in the definitions above have to be replaced by the equivalent 2x2 and 4x4 types.

5.7 Rotations

Matrices are often used in graphics applications to implement transformations like translations and rotations. Because of that the math lib also defines a number of functions to calculate some commonly used rotation matrices.

These rotation function have their own group, *Math.Rotate*

```
group Math.Rotate

function X(float angle) out matrix3
function Y(float angle) out matrix3
function Z(float angle) out matrix3
function Axis(xyz u, float angle) out matrix3

end
```

The first three functions return a 3x3 matrix for the rotation over a given angle around one of the main axes. The last function calculates the rotation matrix for a rotation around a random axis over a given angle, using the well-known Rodrigues rotation formula.

A 3x3 matrix can be casted to a 4x4 matrix. In that case a row and a column is added to the three by three matrix, with all elements set to zero, except the diagonal element c44, which is set to 1.

```
matrix3 M3 = [      a, b, c,
                    d, e, f,
                    g, h, i ]

matrix4 M4 = <matrix4>M3

M4 is then   [      a, b, c, 0,
                    d, e, f, 0,
                    g, h, i, 0,
                    0, 0, 0, 1 ]
```

5.8 Quaternions

Quaternions are vectors with four components, floats or doubles, that have some specific operations defined for them. Applications of quaternions include computer graphics where they are often used to implement rotations.

The quaternion is derived from the built-in vector type vec4f for float components and vec4d for double components. Below we give the definitions in the math lib for the float quaternion. The same defintions exist also for the double quaternion, *quat_d*.

In the literature the components of the quaternion are often referred to as *a,b,c* and *d*, therefore we rename the vec4f components also in the definition. The type name *quaternion* is also introduced as an alternative for *quat_f*.

```
type quat_f is vec4f with

          rename c1 to a
          rename c2 to b
          rename c3 to c
          rename c4 to d

          function power(float exponent) out quat_f
          function conjugate out quat_f
          function norm_squared out float
          function inverse out quat_f

end

-- some specific quaternion operations are part of the CellSpeak bytecode

operator *(quat_f Q, quat_f P) out quat_f
operator *(quat_f Q, xyzw_f V) out xyzw_f

-- We use quat and quaternion as an alternative for the float version

type quaternion is quat_f
type quat is quat_f
```

There are two multiplication operators defined for quaternions: the first one is to multiply two quaternions resulting in another quaternion, and the second is to multiply a quaternion and a 4-vector (xyzw), resulting in another vector.

6 Strings library

The built-in type from which all string types are derived in CellSpeak is the array of bytes or *byte[].*

The type *zero terminated* or *zt* in short, is an array of bytes where the end of the content is indicated by a zero byte. Irrespective of the encoding of the characters in the array this allows to define certain functions such as *length* or *append*.

Three other types are derived from the *zt* type, based on the encoding of the characters:

- ascii: each character is represented by a seven-bit ascii code
- ansi: each character is represented by an eight-bit code whereby the lower codes 0-127 correspond to the ascii codes and the higher codes correspond to a particular interpretation determined by the OS or convention. On Windows systems for example, higher values (128-255) usually refer to the Windows code page CP-1252
- utf8: this is an encoding of Unicode code points into an encoding that is compatible with the ascii code. However characters are not limited to single byte. To represent all Unicode code points, a utf8 encoded code point can take up to 4 bytes. utf8 is currently by far the most popular character encoding.

Because strings are arrays of bytes, all functions, operators and directives that are available for arrays of bytes are available for strings, most notably the *sel, nel* and *lel* operators. Note however that these operators only know about the size of the array and not the length of the string:

```
utf8 Name[40]
Name = “Zweig”
int a = nel Name       -- a is 40
int b = Name.Lenght()  -- b is 5
```

6.1 Are string operations safe ?

6.2 Zero terminated strings

As already mentioned, all string types are derived from the zero terminated or *zt* type. The *zt* type is not often used itself, but rather through its derived types like ascii, ansi and utf8.

The *zt* type is a defined as an array of bytes where a zero byte indicates the end of the content of the array. As for all arrays, the bytes in the string are indexed by an integer starting at 0.

The following methods are defined on the type:

```
group Strings

const failed = -1

type zt is byte[] with

          function len out int
          function lenz out int
          function last out int
          function find(byte b) out int
          function find(zt s) out int
          function fill(byte x)
          function append(zt String) out zt

end

function compare(zt a, zt b) out int
```

When a method can fail it returns *failed*, i.e. the integer value -1.

6.2.1 zt.len

	zt.len()
description	returns the length of a string.
input	-
output	int
remarks	-
example	q.len() where q is “abc” returns 3

6.2.2 zt.lenz

	zt.lenz()
description	returns the length of a string including the terminating 0
input	-

output	int
remarks	This function can be used to allocate strings of the same length of an existing string: sel A[B.lenz()]
example	q.lenz() where q is “abc” returns 4

6.2.3 zt.last

	zt.last()
description	returns the position of the last non-zero byte in a string
input	-
output	int
remarks	-
example	q.last() where q is “abc” returns 2

6.2.4 zt.find

	zt.find(byte B)
description	returns the first location of the byte in the string
input	the byte to locate
output	the position of the byte, *failed* if not found
remarks	
example	q.find(0x’b’) where q is ”abc” returns 1

6.2.5 zt.find

	zt.find(zt String)
description	returns the position of the first occurrence of the substring
input	zt String: the substring to find
output	position of the first character of the substring, *failed* if not found
remarks	
example	q.find(“bc”) where q is “abc” returns 1

6.2.6 zt.fill

	zt.fill(byte B)
description	fills the string with the byte, the last byte is set to 0
input	byte B: the byte to fill the string with
output	-
remarks	-
example	q.fill(0x'a') where q is an array of 5 bytes results in the string "aaaa". The fifth byte is set to 0.

6.2.7 zt.append

	zt.append(zt String)
description	appends a string
input	zt String: the string to append
output	the complete string is returned so the function can be used in expressions.
remarks	if the string to which the function is applied is not big enough to contain the complete string, the string will be re-allocated.
example	q.append("def") where a is "abc" results in q being set to "abcdef"

6.2.8 compare(zt, zt)

	compare(zt S1, zt S2)
description	compares two strings
input	the two strings to be compared
output	int – 0 when the two strings are equal and a-b when the two strings are different, where a and b are the first characters in respectively the first and the second string where the strings differ.
remarks	compare is equivalent to strcmp in c. Note that this function is redefined for utf8 encoded strings, because in this form it does not produce valid results.
example	compare("atom", "atoll") returns 1 because m – l = ascii = 1

6.3 Operators on zero-terminated strings

Instead of using functions, *zt* strings can be compared using the familiar comparison operators:

```
== or is
!= or is not
>
>=
<
<=
```

The result of each comparison is a Boolean value, true or false. The comparison is done as explained for the *compare* function.

Two strings can also be appended using the + operator:

```
zt a, b
a = “this is”
b = “ the total string”

a = a + b

results in a becoming “this is the total string”
```

6.4 Ascii strings

Ascii strings are zero terminated strings for which the

```
type ascii is zt with     -- ascii inherits from zero-terminated

          function upper
          function lower

end

-- We also define functions that do not alter the input string
function upper(ascii String) out ascii
function lower(ascii String) out ascii
```

6.4.1 ascii.upper()

	ascii.upper()
description	converts an ascii string to all uppercase characters
input	-
output	-
remarks	-
example	q.upper() where q is “aBc” results in q being set to “ABC”

6.4.2 ascii.lower()

	ascii.lower()
description	converts an ascii string to all lowercase characters
input	-
output	-
remarks	-
example	q.lower() where q is “aBc” results in q being set to “abc”

6.4.3 upper(ascii String)

	upper(ascii String)
description	converts an ascii string to all uppercase characters

input	the ascii string to convert
output	the converted ascii string
remarks	-
example	upper(q) where a is “abc” results in q being set to “ABC”

6.4.4 lower(ascii String)

	lower(ascii String)
description	converts an ascii string to all uppercase characters
input	the ascii string to convert
output	the converted ascii string
remarks	-
example	upper(q) where a is “abc” results in q being set to “ABC”

6.4.5 ascii constants

The ascii characters up to 0x20 (space) are also known by their code-name (ACK, CR, LF etc) in order to be able to refer to the ascii-characters using their code name, the ascii characters are also defined as constants in the group ASCII. There are also constants defined for the non-alphanumerical ascii codes.

```
-.
We define a group ASCII to give names to ascii bytes.
It can make code more readable that deals with ascii.

There is no char type in CellSpeak - because a char type only has meaning
in the context of encoding. There is a byte type. Byte constants can be
given names - as below - or can be written as 0x00 (two hex digits).
Byte constants can also be written as 0x'n' where n is any printable ascii
character, eg 0x'a' or 0x' ' or 0x'?' etc.
.-
group ASCII

const byte NUL            = 0x00 --    Null char
const byte SOH            = 0x01 --    Start of Heading
const byte STX            = 0x02 --    Start of Text
const byte ETX            = 0x03 --    End of Text
const byte EOT            = 0x04 --    End of Transmission
const byte ENQ            = 0x05 --    Enquiry
const byte ACK            = 0x06 --    Acknowledgment
const byte BEL            = 0x07 --    Bell
const byte BS             = 0x08 --    Back Space
const byte HT             = 0x09 --    Horizontal Tab
```

```
const byte LF             = 0x0a --    Line Feed
const byte VT             = 0x0b --    Vertical Tab
const byte FF             = 0x0c --    Form Feed
const byte CR             = 0x0d --    Carriage Return
const byte SO             = 0x0e --    Shift Out / X-On
const byte SI             = 0x0f --    Shift In / X-Off
const byte DLE            = 0x10 --    Data Line Escape
const byte DC1            = 0x11 --    Device Control 1 (oft. XON)
const byte DC2            = 0x12 --    Device Control 2
const byte DC3            = 0x13 --    Device Control 3 (oft. XOFF)
const byte DC4            = 0x14 --    Device Control 4
const byte NAK            = 0x15 --    Negative Acknowledgement
const byte SYN            = 0x16 --    Synchronous Idle
const byte ETB            = 0x17 --    End of Transmit Block
const byte CAN            = 0x18 --    Cancel
const byte EM             = 0x19 --    End of Medium
const byte SUB            = 0x1a --    Substitute
const byte ESC            = 0x1b --    Escape
const byte FS             = 0x1c --    File Separator
const byte GS             = 0x1d --    Group Separator
const byte RS             = 0x1e --    Record Separator
const byte US             = 0x1f --    Unit Separator

const byte Space          = 0x20
const byte Exclamation    = 0x21
const byte DoubleQuote    = 0x22
const byte Hash           = 0x23
const byte Dollar         = 0x24
const byte Percent        = 0x25
const byte Ampersand      = 0x26
const byte SingleQuote    = 0x27
const byte RoundOpen      = 0x28
const byte RoundClose     = 0x29
const byte Star           = 0x2a
const byte Plus           = 0x2b
const byte Comma          = 0x2c
const byte Hyphen         = 0x2d
const byte Period         = 0x2e
const byte Slash          = 0x2f

const byte Colon          = 0x3a
const byte SemiColon      = 0x3b
const byte LessThen       = 0x3c
const byte Equals         = 0x3d
const byte BiggerThen     = 0x3e
const byte QuestionMark   = 0x3f
const byte At             = 0x40
```

6.5 ANSI encoding

Ansi encoding is compatible with ascii encoding – all ascii encoded strings are ansi encoded. The ansi encoding is a name used for an encoding format used on windows systems where the codes from 128 to 255 map onto an additional set of mostly accented characters (windows cp-1252).

There are no specific functions for ansi encode strings.

6.6 utf8 strings

UTF-8 encoding is compatible with ascii encoding , ie all ascii encoded strings are utf8 encoded.

UTF-8 encoding is a variable length encoding for Unicode code points. A Unicode code point can be encoded in one, two, three or four bytes. So in utf8 a single byte does not necessarily correspond to a single character representation. A Unicode code point is identified by a 32-bit value.

6.6.1 utf8.append

	utf8.append(uncode CodePoint)
description	appends a code point to a utf8 string
input	Unicode CodePoint: the code point to append
output	-
remarks	Unicode code points are limited to 21 bits. An invalid code point will be converted to the following three bytes: EFBFBD – the utf8 encoding for the replacement character.

example	`utf8 Greek = "Greek alphabet: "`
	`for i=0 to 24 do`
	`  Greek.append(0x000003b1 + i)`
	`end`
	
	results in Greek = “Greek alphabet: αβγδεζηθικλμνξοπρςστυφχψω”

7 The structures library

A programming language offers a basic set of types and operations to build higher level constructs. Very often we find that across many types of applications, we are re-using the same types of constructs over and over again, examples are hash-tables, linked lists etc. Despite the fact that these constructs are recurring design patterns, it would be unpractical to add these constructs as built-in types of the language, because they are relatively complex types, and there is often a need for specific functions, operators and methods that require a great deal of flexibility in the definition of the construct.

The combination of inheritance and templates in CellSpeak, allows to build flexible re-usable constructs that are applicable in many situations. The structures library defines a number of these base-types and templates, that can be used to derive the specific types for a given application, or that can be used as starting points to build new types and templates for a particular type.

The structures are part of the group *Structures*. Often a specific structure also has its own group, like the double linked lists : *Structures.DoubleLinked*.

7.1 Double Linked List

Each element in a double linked list, except the first and the last, is linked to the element before and the element after itself.

The linked list in the structures library consists of two parts: a simple record definition that contains the two pointers *previous* and *next* and some simple functions to add and remove an element, and a linked list template, based on the record definition – or any type derived from it – that offers some additional functionality.

In many cases deriving a record from the base record with the pointers is all that is needed to have the basics of a linked list – additional functions can then be added to manipulate the list, e.g. to find elements in the list based on the values of the fields in the derived record etc.

But because some of these actions are also common, the library also contains a template for the linked list consisting of the *head* of the linked list and some basic functions like *Push* and *Pop.*

In the following example we derive a record from the double linked record and add some fields and methods:

```
type TimerRequest is Structures.DoubleLinked.Record with

          cell            Requestor
          word32          Interval
          utf8            Message
          function Find(cell CellId) out TimerRequest is
          end

end
```

If we want to use the functionality offered by the linked list template for this type we can then instantiate the template with this type:

```
use template Structures.DoubleLinked.List for BaseTimerList, TimerRequest
```

The first parameter is the name of the linked list, and the second parameter is the type that is used in the linked list.

Often we will want to add some additional functions to the type(s) that were created by instantiating the templet. This can be done in the usual way:

```
type TimerList is BaseTimerList with

          … additional methods and fields here

end
```

The template can also be used of course as a basis to build a new template more suited to the types or style of the application being build.

7.1.1 Structures.DoubleLinked.Record

The base record of a double linked list is defined in the group *Structures.DoubleLinked* as follows:

```
type Record is record

          Record.ptr      Previous
          Record.ptr      Next

          function InsertAfter( out Record R)
          function InsertBefore( out Record R)
          function Remove

end
```

7.1.2 Record.InsertAfter(out Record)

description	**inserts this record after the parameter record**
input	*Record*: the record after which this record has to be inserted.
output	-
remarks	The parameter has to be an *out* parameter, because the *Next/Previous* fields of the record can be changed.
example	GreenRecord.InsertAfter(BlueRecord) : the GreenRecord will be inserted after the BlueRecord.

7.1.3 Record.InsertBefore(out Record)

description	**inserts this record after the parameter record**
input	*Record*: the record before which this record has to be inserted.
output	-
remarks	The parameter has to be an *out* parameter, because the *Next/Previous* fields of the record can be changed.
example	GreenRecord.InsertBefore(BlueRecord) : the GreenRecord will be inserted before the BlueRecord.

7.1.4 Record.Remove

description	**removes the record from the linked list**
input	-
output	-
remarks	The record is not deleted, only removed from the linked list. Works also if one or both of the *Next/Previous* pointers is null.

example	GreenRecord.Remove() – the record will be removed from the linked list is is in.

7.1.5 Structures.DoubleLinked.List

The template is instantiated as follows:

```
use template Structures.DoubleLinked.List for NewTypeName, RecordType
```

Where NewTypeName is the new type name of the double linked list and *RecordType* is the record used in the linked list. The only condition for the record type is that it must have two pointer fields to the same type, named *Previous* and *Next.*

```
type NewTypeName is record

          RecordType.ptr Head

          function Push(out RecordType R)
          function Pop() out RecordType
          function Remove(RecordType R)
          function New out RecordType

end
```

The linked list is a record type with one field, the pointer to the first record in the list, and four methods.

7.1.6 List.Push(RecordType)

description	**Adds a record at the start of the list**
input	*RecordType:* the record to push on the list
output	-
remarks	-
example	-

7.1.7 List.Pop()

description	**Take the first record from the list**
input	-
output	The pointer to the first record from the list
remarks	The record is not deleted
example	-

7.1.8 List.Remove(RecordType)

description	**Removes the record from the list**
input	*RecordType*: the record to remove
output	-
remarks	The record is not deleted.
example	List.Remove(P)

7.1.9 List.New()

description	**Returns a new record.**
input	-
output	*RecordType*
remarks	-
example	-

7.2 Hash Table

The hash table structure allows to create a table of records that can easily be retrieved based on the content of a string field, for example a name. In order to store a record a key is calculated from the string field, the *hash key*, and that key is used to store and retrieve the field from a structure of tables.

The hash table in the structures library is a template where a number of items have to passed when the template is instantiated.

```
template HashTable for NewType, Type, Field, Size
```

- *NewType* is the name for the newly created hash table type
- *Type* is the type of the record that is to be stored in the hash table
- *Field* is the name of the string field in the record that will be used to calculate the hash key
- *Size* is the size in bits of the hash key

When the type has been created, it has to be initialized once by calling the method *init().* The init method will create a table of 2^size entries (e.g if the hash key is 8 bits the table size is 256).

Because a hash key is often not unique, the hash table will create additional tables for 'backstore' as required. The type has the following methods that can be used:

```
type NewType is record
          function Add(utf8 Text) out Type
          function Find(utf8 Text) out Type
end
```

7.2.1 HasTable.Add(utf8 String) out Type

description	**adds a new record to the hash table**
input	The key-string
output	The pointer to the record in the table
remarks	if a record with the same key-string is already in the table, the function returns null
example	-

7.2.2 HashTable.Find(utf8 String) out Type

description	**returns the record that corresponds with the key-string**
input	the key-string
output	Pointer to the record
remarks	If a record with that key-string does not exist, the function returns null

example	-

8 File handling

The file handling library contains a list of functions to read and write files from a local file system.

8.1 Overview

```
group System.File

-- System.File.Record - use this if you need to open a file that you need to
access multiple times.
type Record is record

          FileClass      WinFile

          function Open(utf8 Name, utf8 Mode) out bool
          function Close
          function GetSize out int
          function Read(out byte[] Data ) out int
          function Read(word Max) out byte[]
          function Read out byte[]
          function Write(byte[] Data) out bool

end

-- The following functions allow to read or write a file in one go (open read
close in one function)
function Read(utf8 Name, word Max) out (int Error, byte[] Data)
function Read(utf8 Name) out (int Error, byte[] Data)
function Write(utf8 Name, byte[] Data) out (int Error)

group System.Directory

function GetFileList(utf8 Directory) out utf8[]
```

9 Windows Library

The windows library contains the functions and datatypes that are closely linked to the services offered by the OS. The library imports a C library that contains the classes that can be used in directly in CellSpeak or around which new CellSpeak types can be built.

9.1 Overview

The group *Windows* imports four C++ classes

```
group Windows is WindowsLib

lib class BasicWindowClass
lib class CanvasWindowClass extends BasicWindowClass
lib class ClockClass
lib class FileClass
```