|  |
| --- |
| Walker Corp |
| Walker OS |
| A comprehensive guide |

|  |
| --- |
| Jonathan van de Veen  5-6-2022 |

Contents

[Console 2](#_Toc117075196)

[Console output 2](#_Toc117075197)

[Console input 2](#_Toc117075198)

[Walk programming language 3](#_Toc117075199)

[Syntax 3](#_Toc117075200)

[Variable names 3](#_Toc117075201)

[Variable declarations 3](#_Toc117075202)

[WalkerFS 4](#_Toc117075203)

[Collections 4](#_Toc117075204)

[Files 4](#_Toc117075205)

# Console

## Console output

To get output from any process to the console, Walker OS uses an OutputBuffer. This is implemented as a string array. The OS handles this string array as a stack.

## Console input

Console input uses an input buffer. This is implemented as a string array.

# Walk programming language

The Walk programming language is a first-class citizen in the Walker OS. The default console takes Walk statements and allows you to execute complete Walk programs through an interpreter.

## Syntax

Each statement ends with ;

Functions are called by specifying their name followed by () in between parameters can be specified if needed. Parameters are separated by a ,

Functions are defined by specifying their name followed by their parameters between (). The code block starts with { and ends with }.

## Variable names

Variable names cannot begin with an underscore (\_). Names starting with an underscore are reserved for the interpreter.

## Variable declarations

Variables do not need a declaration. When first encountered the interpreter will make sure the variable is available.

## Functions

Functions are written as follows:

fun(arg1, arg2) { … }

where fun is de name of the function and args1 and args2 are arguments for the function. The code for the function is placed between { and }.

To return a value from a function you assign a value to a variable with the same name as the function.

# WalkerFS

This is the storage system used within WalkerOS. It uses files as the atomic unit of storage. Files are organized in collections.

## Collections

Collections are somewhat like traditional folders. The main difference is that they are not organized in a tree, but in a graph. This means you can add collections to other collections and even create loops in these structures. The graph structure also applies to the way files are placed in collections. Files are **not** limited to be in one single collection.

The following rules apply:

* Collections must be part of at least one other collection, except for the root collection
* Files must be part of at least one collection

Collections have a name, an ACL and an id.

When creating a new collection, you specify the name and a parent collection. The id is assigned by the OS.

By default, the collection will inherit the ACL. If an updated ACL is needed, it can be specified when creating the collection or updated afterwards.

## Files

Files have a name, an optional ACL and an id.

When creating a ne file, you specify the name, and a parent collection. The id is assigned by the OS.

By default, files do not need an ACL. In this case the ACL of the collection is used.

# OS internals

## Notes on Executionspaces

Code is always executed in an executionspace. Each process gets its own executionspace. Each executionspace has its own stack. This leads to built-in separation between processes and users. Therefor executionspaces also have a user attached to them.

Whenever code starts a new process, if effectively creates a new executionspace. This new exectionspace has a parent which is the space that caused the creation.

When a user starts a console session, this creates an executionspace to be used for code executed from the console. This means code executed in the console shares a stack as well. Executionspaces created for console sessions are root executionspaces.

Executionspaces also contain a queue of instructions that are executed. Whenever the queue is empty, the executionspace is destroyed, unless they are root executionspaces.