Justina library
User Manual

2024, Herwig Taveirne

The library is intended to work with 32-bit boards using the SAMD architecture , the Arduino nano RP2040 and Arduino nano ESP32 boards.

See GitHub for more information and documentation: https://github.com/Herwig9820/Justina_interpreter

Developer contact: herwig.taveirne@gmail.com

Publish date: 29/05/2024

Table of Contents

1 Introduction

Justina is both an easy-to-use programming language for Arduino and a capable interpreter.

It has been developed and built around a few objectives. On top of the list: simplicity for the user. Justina is a structured language, but it's non-object oriented (as opposed to the powerful but more complex c++ language). It has some similarities with Basic, a language that has been around for quite some time. But (and this was, of course, a main objective) it was built with Arduino in mind - more specifically, 32-bit Arduino's: boards with a SAMD processor (like the nano 33 IoT), nano ESP32 boards and nano RP2040 boards.

Justina does not impose any requirements or restrictions related to hardware (pin assignments, interrupts, timers,... - it does not use any), nor does it need to have any knowledge about it for proper operation.

The Justina syntax has been kept as simple as possible. A program consists of statements. A statement either consists of

- a single expression (always yielding a result).
- a command, starting with a keyword, optionally followed by a list of expressions (such a statement is called a command, because it 'does' something without actually calculating a result)

Because Justina is an interpreted language, a Justina program is not compiled into machine language but it is parsed into so called tokens before execution. Parsing is a fast process, which makes Justina the ideal tool for quick prototyping. Once it is installed as an Arduino library, call Justina from within an Arduino c++ program and you will have the Justina interpreter ready to receive commands, evaluate expressions and execute Justina programs.

As an added advantage, you can enter statements directly in the command line of the Arduino IDE (the Serial monitor by default, a TCP IP client, ...) and they will immediately get executed, without any programming.

Example

In this first example, we will first set the console display width for calculation results to 40 characters wide (by default, it's set to 64) and set the angle mode to Degrees. We'll then define Arduino pin 17 as an output and write a HIGH value to the pin. Finally, we'll calculate the cosine of 60°.

In the command line of the Arduino IDE Serial Monitor, type these three lines (each time followed by ENTER):

```
dispWidth 40; angleMode DEGREES;
pinMode( 17, OUTPUT); digitalWrite(17, HIGH);
cos(60);
```

Statements typed are echoed after the Justina prompt ("Justina>") and executed. Multiple statements can be entered at the same time, separated by semicolons.

```
Justina> dispWidth 40; angleMode DEGREES              → 2 commands
Justina> pinMode(17, OUTPUT); digitalWrite(17, HIGH)  → 2 expressions
                                             1             result
Justina> cos(60)                                      → expression
                                         0.50              result
Justina>
```

The result of the last expression entered in the command line is printed on the next line. In this example: both *digitalWrite()* and *cos()* are functions, *digitalWrite* returning the value written to the pin (1 is the value of predefined constant HIGH). If the anode (+) of a led is connected to pin 17, and, via a proper resistor, the cathode (-) is connected to GROUND, the led will be ON. Commands do not return any result.

A few highlights

- More than 250 built-in functions, commands and operators, 70+ predefined symbolic constants.
- More than 30 functions directly targeting Arduino IO ports and memory, including some new.
- Extended operator set includes relational, logical, bitwise operators, compound assignment operators, pre- and postfix increment operators.
- Two angle modes: radians and degrees.
- Scalar and array variables.
- Floating-point, integer and string data types.
- Perform integer arithmetic and bitwise operations in decimal or hexadecimal number format.
- Display settings define how to display calculation results: output width, number of digits / decimals to display, alignment, base (decimal, hex), …
- Input and output: Justina reads data from / writes data to multiple input and output devices (connected via Serial, TCP IP, SPI, I2C...). You can even switch the console from the default (typically Serial) to another input or output device (for instance, switch console output to an OLED screen).

```
Justina> listFiles

SD card: files (name, size in bytes):
System Volume Information/
  WPSettings.dat             12
  IndexerVolumeGuid          76
data001.log                  13034
data002.log                  13034
photo.JPG                    2750731
Justina/
  images/
    Jus_icon.jpg             1464
    Jus_logo.jpg             13034
  start.jus                  2932
  web_calc.jus               19387
  web_swit.jus               8458
```

List of SD card files, including Justina programs (.jus)

- With an SD card breakout board connected via SPI, Justina creates, reads and writes SD card files etc.
- In Justina, input and output commands work with argument lists: for instance, with only one statement, you can read a properly formatted text line from a terminal or an SD card file and parse its contents into a series of variables.

Programming

- Write program functions with mandatory and optional parameters, accepting scalar and array arguments. When calling a function, variables (including arrays) are passed by reference. Constants and results of expressions are passed by value.
- Variables or constants declared within a program are either global (accessible throughout the Justina program), local (accessible within a Justina function) or static (accessible within one Justina function, value preserved between calls)
- Variables not declared within a program but by a user from the command line, are called user variables (or user constants)

- Programs have access to user variables and users have access to global program variables (from the command line. User variables preserve their values when a program is cleared or another program is loaded.
- Parsing and execution errors are clearly indicated, with error numbers identifying the nature of the error.
- Error trapping: if enabled, an error will not terminate a program, instead the error can be handled in code (either in the procedure where the error occurred or in a 'caller' procedure). It's even possible to trap an error in the command line

Program editing

You can use any text editor to write and edit your programs. But you might consider using Notepad++ as text editor, because a specific 'User Defined Language' (UDL) file for Justina is available in the Justina library, providing Justina syntax highlighting as shown in the example below. See Appendix F: *Installing Notepad++ and the Justina language extension*.

```
function writeRecords();

    var testFile = 0;
    if (testFile = fileNum("people.txt")) > 0; close (testFile); end;

    // using printList instead of printLine:
    // - strings will be printed with surrounding quotes (safe for string
    // - numbers will be written with full accuracy
    testFile = open("people.txt", WRITE | TRUNC | NEW_OK);
    printList testFile, "John", "blue", "gray", 172, 78.3, 23;
    printList testFile, "Percy", "brown", "brown", 168, 75.7, 58;
    printList testFile, "Tracy", "green", "gray", 175, 58.4, 42;
    printList testFile, "Basil", "blue", "red", 177, 81.2, 51;
    printList testFile, "Caroline", "green";
    printList testFile, "Irene", "brown", "gray", 169, 61.8, 75;
    printList testFile, "no\\na\"me";
    printList testFile, "Charles", "green", "blond", 172, 79.3, 48;

    close(testFile);

    // read back and print to console:
    testFile = open("people.txt", READ);
    while (available(testFile) > 0);
        cout readLine(testFile);
    end;
    close(testFile);
end;
```

*Excerpt of a **Justina** program, edited in Notepad++ with the Justina language extension installed. Distinct colors highlight different language elements.*

Debugging

- When a program is stopped (either by execution of the '**stop'** command, by user intervention or by an active breakpoint) debug mode is entered. You can then single step the program, execute statements until the end of a loop, a next breakpoint…
- Breakpoints can be activated based on an optional trigger expression or a hit count. You can also include a list of 'view expressions' for each breakpoint, and Justina will automatically trace specific variables or even expressions, letting you watch their values change, as you single step through the program or a breakpoint is hit.

```
Justina> listBP

Breakpoints are currently ON

source   enabled   view &
  line             trigger
------   -------   -------
    77      x      view : subTotal, total;
                   trigger: (n= < 10) || (n = last);
   104      x      view : temp - refTemp;
                   trigger: speed < 50;
   121      x      view : fx;
                   hit count: 100 (current is 0)

Justina>
```

List all defined breakpoints for a program

While a procedure is stopped in debug mode, you can also manually review the procedure's local and static variable contents or view the call stack.

Integration with c++

1. If enabled, system callbacks allow the Arduino program to perform periodic housekeeping tasks beyond the control of Justina (e.g., maintaining a TCP connection, producing a beep when an error is encountered, aborting, or stopping a Justina program...). For that purpose, a set of system flags passes information back and forth between the main Arduino program and Justina at regular intervals (without the need for interrupts).

   ➲ See Appendix D: *Running background tasks: system callbacks*.

2. Built-in Justina functionality can be extended by writing specific functions in c++. Such '*user c++*' functions include time-critical user routines, functions targeting specific hardware, functions extending functionality in a specific domain, etc. These functions are then 'registered' with Justina and given an alias.
   From then onward, these C++ functions can be called just like any other Justina function, with the same syntax, using the alias as function name and passing scalar or array variables as arguments.
   You can even *write complete Justina user c++ libraries*, if desired.

   ➲ See Appendix E: *calling user c++ functions*.

2 Getting started

Start by installing the Justina library, named 'Justina interpreter', from the Arduino library manager.

- In the Arduino IDE, select 'Tools -> Manage Libraries' and filter the library list by "Justina"
- Click 'Install', next to the library named 'Justina'.

Now let's immediately try a small Arduino program. It will simply call Justina and stay there (until we tell it to *return to the calling Arduino program).*

```
#include "Justina.h"

// create Justina_interpreter object with default values
Justina justina;

// ------------------------------
// *   Arduino setup() routine   *
// ------------------------------

void setup() {
    Serial.begin(115200);
    delay(5000);
    // run interpreter (control will stay there until you quit) Justina)
    justina.begin();
}

// -----------------------------
// *   Arduino loop() routine   *
// -----------------------------

void loop() {
    // empty loop()
}
```

A simple Arduino c++ program to launch the Justina interpreter

The Arduino program is provided as a sample sketch in Justina's library 'examples' folder, named 'Justina_easy.ino'.

Arduino IDE: *File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_easy.ino*

Verify the baud rate; the Arduino IDE Serial Monitor should have status 'connected' (we will use this Serial Monitor, for now).

Load and run the sketch. You should see:

```
*************____****_**************_**********
    Justina: JUST an INterpreter for Arduino
    Copyright 2024, Herwig Taveirne
    Version: 1.1.1
***********************************************

Justina> |
```

The 'Justina>' prompt indicates that Justina is currently running. Each time you enter a statement (in the command line), the statement will be echoed after the prompt and subsequently executed.

Let's start by typing in a simple expression in the command line: `3 + 5;` (+ ENTER)

Serial Monitor output:

```
Justina> 3 + 5
                                8
Justina>
```

The result of the expression, '8', is displayed on the next line, right-aligned (output format and alignment can be changed in Justina display settings).

Let's create a user variable now: In the command line, type `var myFirstVar = 10;` (+ ENTER)

Serial Monitor output:

```
Justina> var myFirstVar = 10
Justina>
```

The characters **var** (all lowercase) form a keyword, indicating the start of a command. Commands 'do' something (in this case, creating a variable and optionally initializing it with some constant value) but they don't produce a result, so a result is not printed.

Let's now enter multiple statements together. Just make sure you separate statements with a semicolon.

In the command line, type `3 + 5; 7 + 8; myFirstVar += 12;` (+ ENTER)

Serial Monitor output:

```
Justina> 3 + 5; 7 + 8; myFirstVar += 12
                               22
Justina>
```

As expected, the three expressions are echoed after the prompt, but only one result is printed: the result of the last expression (the initial value of variable a was '10'). Because the first two expression results were not stored in a variable, these results are lost.

The '+=' operator means 'add the result of the expression to the right (12) to the variable on the left'.

You could also have typed `myFirstVar = myFirstVar + 12;`

Finally, let's deliberately produce some errors and see what happens.

Parsing errors

In the command line, type

```
myFirstVar = 20; myFirstVar += 3 + 5 + * 7-2; 20 + 21;
```

Serial Monitor output:

```
Justina>
  3+5+*7-2;
      ^
  Parsing error 1103
Justina>
```

Even before the result of the expression could be calculated, a **parsing** error occurred: the interpreter detected a syntax error in the second expression (the '*' makes no sense there).

Nothing is echoed after the prompt, instead the expression containing the error is printed with a caret symbol indicating the position of the error. Looking up the parsing error message number 1103 in the documentation reveals that an invalid operator was detected.

Execution errors

In the command line, type `123 + asin(-2) + 789;`

Serial monitor output:

```
Justina> 123 + asin(-2) + 789

  123 + asin(-2) + 789
          ^
  Exec error 3100
Justina>
```

As parsing went OK, your input is echoed after the prompt.

But there's still a problem: the domain for the inverse sine function *asin()* is [-1, 1]. So, an **execution** error occurs and the position of the error is shown.

Error number 3100 indicates that an argument is out of range.

A simple loop

In the command line, type `var i; for i = 1, 5; coutLine "line = ", i; end;`

Serial monitor output:

```
Justina> var i; for i = 1, 5; coutLine "line = ", i; end
line = 1
line = 2
line = 3
line = 4
line = 5
Justina>
```

The words **for**, **coutLine** and **end** are all keywords, indicating the start of a command. We will discuss the complete syntax later, but for now:

for and **end** form a loop structure. In this example, they instruct Justina to execute the statements in between 5 times, each time augmenting the value of I by 1.
coutLine ('console out line') prints its arguments to the console and moves to the next line.

Note that command arguments, just as function arguments, are separated by a comma. But the command argument list is not put between parentheses in contrast to function arguments.

Statements (commands or simple expressions) are separated by a semicolon.

Editing and saving your first program

On your computer, in notepad, create a text file with the following text (in next examples we'll switch to notepad++, offering line numbering and Justina syntax highlighting).

```
program myFirstProgram;     // this is a JUSTINA program
var i;                      // this is a global PROGRAM variable
function print5lines():     // this is a function
  for i = 1, 5;             // this is the start of a loop
    coutLine "line = ", I;  // this prints something
  end;                      // this is the end of a loop
  return I ** 2;            // this returns the square of I
end;                        // this the end of a function
```

Save the program under a name, let's say 'myFirst.jus' .

Note: in a Justina program, line comments start with two slash characters. All text starting with '//' until the end of the same line is simply discarded during parsing.
Line comments and multi-line comments will be discussed in chapter 13: *Programming*.

Now, we need to get this program into the Arduino (for the moment, let's assume an SD card reader is not attached to your Arduino, so we cannot get it from there).

Installing a Terminal program on your computer

Unfortunately, we cannot use the Arduino IDE Serial Monitor to send files to the Arduino board (for those developing with Visual Studio and the VisualMicro Arduino IDE: same issue).

Luckily enough, there are a few good free terminal programs out there. The one I prefer is YAT and we will use it throughout this manual. A second one which works quite well is named Tera Term. These terminal programs are freely downloadable on your PC. They allow for serial communication via USB as well as via TCP / IP connections.

In what follows, we'll stick to YAT because it has a couple of nice, useful features.

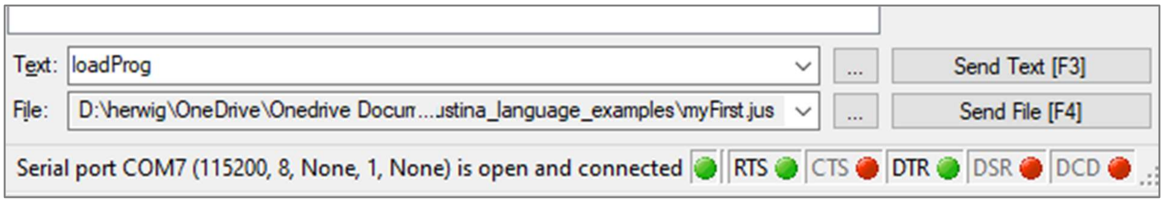
➲ To download, install and setup YAT, please refer to Appendix G: *Installing YAT* .

Assuming that you installed YAT, you can now use YAT as your serial monitor to send Justina statements to your Arduino: when you type a statement in the 'Send Text' textbox and press Enter, you'll see your Arduino's response, as you did in the preceding examples.

Sending a Justina program to your Arduino

In Terminal Settings, verify that "Serial COM port" is selected as I/O type, the correct serial port is selected and the baud rate is set.

Also, at the bottom of the terminal window, check that indicators RTS (request to send) and DTR (data terminal ready) both show a green light. If not (showing red), click the indicators to set RTS and DTR ON (indicators should switch to green).



YAT terminal window (lower part)

To connect, click *Terminal -> open/start* or click the green 'open/start terminal' button.

⚠ Remember to close the (Arduino, MS Visual Studio, ...) IDE Serial Monitor before connecting the Terminal app to your Arduino.

You're ready to load your first program now. Referring to the figure above:

- Using the button with 3 dots to the left of YAT key 'send file', select the program you just saved (note that this file is also available as part of the Justina language example programs, in library folder *'libraries\Justina_interpreter\extras\Justina_language_examples\'*, in your Arduino sketchbook location).
- Send the command **loadProg** to Arduino (type it in the 'Send text' textbox and press ENTER or click button 'Send Text'). This will instruct Justina to start waiting for a Justina program, listening to the 'console input' device (Serial, as defined in the Arduino program that started Justina – see chapter 2: *Getting started*).
- Send the file you just selected to Arduino (button 'Send File')

Notes

- Command **loadprog** times out after 15 seconds if it's not followed by a program.
- You can load a Justina program from any input device, not only the device defined as 'console' - for instance a TCP input stream, an SD card file (if connected) .... We'll discuss that in chapter 12*: Other functions and commands.*

If Justina returns an error code: check your program (text file) and correct any typing errors.

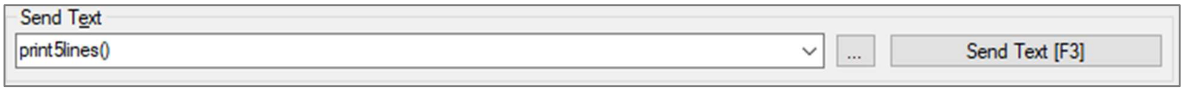
If all is OK, YAT Terminal output will be:

```
Justina> loadProg

Waiting for program...
Receiving and parsing program... please wait

Program parsed without errors. 0 % of program memory used (84 of 6

Justina>
```

Congratulations ! You just loaded your first program. It has been parsed and is ready for execution. Time to run it !

Now, execute function print5lines (it was defined in program myFirstProgram, in file 'myFirst.jus'):

Type `print5lines();` (+ ENTER, or click button Send Text) :



YAT terminal output:

```
Justina> print5lines()
line = 1
line = 2
line = 3
line = 4
line = 5
                              36

Justina>
```

Value 36 is the result returned by the program.

Printing calculation results can be switched off if desired.

Conventions used in this manual

Justina commands are printed in **bold**

Built-in functions are printed in *italic*.

Square brackets ([ ]) indicate optional parts of an argument list

Single quotes (' ') are used for clarification only, and are not part of commands, functions or expressions.

Sample code lines are shown with a fixed spacing font, with a gray background.

Justina console output is shown in a light-colored background.

Examples

const name1 = literal1 [, name2 = literal2, name3 = literal3...] ;

digitalWrite (LED_BUILTIN, HIGH);

```
var n=0, i=0, fact=0;                   // init as integer
n=1; fact=1; for i =2, n; fact=fact * i; end;     // 1!
n=4; fact=1; for i =2, n; fact=fact * i; end;     // 4!
n=6; fact=1; for i =2, n; fact=fact * i; end;     // 6!
```

```
intFmt 8, DEC
Justina> 53
                                          00000053

Justina> intFmt 0
Justina> 123456
                                            123456
```

3 Statements: commands and expressions

A statement consists of either a single expression, or a Justina command.

Multiple statements entered together (on the command line or in a program) must be separated by a semicolon character: *statement ; statement ; statement*...

Expressions

An expression is anything that consists of functions with arguments and operators acting upon operands (function arguments and operands can be expressions themselves). Expressions always yield a result, that is, expressions are evaluated and make available a result.

Examples:

```
3 + sqrt( 5 ) ;
name = firstName + " " + lastName ;
```

(name, firstName and LastName must be declared as variables)

Commands

A command always starts with a **keyword**, optionally followed by a list of expressions, being the arguments of the command. Commands 'do' something (for example declaring and initializing a variable), but they do not return a result.

If a command has arguments, at least one space must separate the command from the first argument.

Note that the argument list, if present, is not put within parentheses. Expressions used as command arguments are separated by a comma, just like function arguments.

Command syntax:

keyword [expression [, expression, expression, expression]...] ;

Examples

```
stopSD ;
copyFile "source.txt", "myCopy.txt" ;
var myName = "John", total = 0 ;
cout 3 + 5 ;
```

In Justina, all identifier names (built-in command and function names, names of user-written functions, variable names, ...) follow the same naming convention: names must start with a letter from a to z (or A to Z), and may be followed by a sequence of letters, digits and underscore characters. The maximum name length is 20.

⚠ In Justina, all identifier names are case sensitive!

4 Data types

Justina works with 3 types of values: signed integers, floating point numbers and variable-length strings.

Signed integers (called 'integers' from this point on) are implemented as c++ 32-bit signed integers; floating point numbers (also called 'floats' from here on) as c++ 32-bit floating-point numbers and Justina strings as c++ variable-length char array heap objects.

Integers and floats are two distinct data types with a different internal representation.

Integers are perfect for loop counters, working with binary numbers, logical and bitwise operators.

Integer and float literals

Any sequence of characters recognized as a number, but without a decimal point or an exponent, will be interpreted as an integer, otherwise the character sequence will be interpreted as a floating-point number.

Integers are perfect to work with Boolean and bitwise operators, or to perform binary arithmetic (discussed in chapter 7: *Operators*)

Integer numbers can be typed in binary or hexadecimal format as well, by preceding the number by a prefix. 0b or 0B indicates binary, 0x or 0X means hexadecimal.

enter	`123;`	integer
	`123.;`	floating point number
	`12e2;`	floating point number: the 'e' is interpreted as 'exponent.'
	`0x12e2;`	integer (base 10 number 4834): the 'e' is a hexadecimal digit.

String literals

Any sequence of characters typed or read and delimited by double quotes (").

Use escape sequences to include special characters as part of a string. An escape sequence consists of a backslash (' \ ') character followed by another character. Four escape sequences are available:

\\	Add a backslash character to a string
\"	Add a double quote character to a string instead of interpreting it as a string delimiter)
\r	Add a 'carriage return' control character (Ascii code 0x0D) to a string
\n	Add a 'line feed' control character (Ascii code 0x0A) to a string

Good to know: in Justina, ***empty strings*** ("") do not create a heap object, which helps in conserving memory.

enter	`"abc";`	store a string containing 3 characters: abc
	`"\"ab\\cd\"ef";`	store a string containing 8 characters: ab\cd"ef
	`"";`	empty string (does not need character storage for the string)
	`"line 1\r\nline2"`	same as `"line 1" + line() + "line2"`

Function `line()` in the last example is a Justina function returning a 2-character string with a CRLF (carriage return line feed sequence.

5 The console

From the perspective of the user, the console is the input/output device sending commands to Justina and displaying system output. Justina looks at it from the other side: Justina receives commands from the console and sends output to it. Right now, the console has been set to the device connected to the 'Serial' stream. Later, we will see how to change the console, e.g., to a device connected to a TCP/IP terminal, an OLED or LCD display etc.

Output sent to the console includes calculation results, echo of user input, error messages etc.

☞ *Note that input/output is not restricted to the console: several commands are available to read data from and send data to any available input/output channel or SD card file (if an SD card board is connected). And as mentioned, it's even possible to change the console itself to another I/O device.*

The following commands allow you to change the way data is displayed:

dispWidth width ;	Changes the display width (for printing calculation results). Minimum is 0, maximum is 255 - even if larger values are entered. Justina will try to fit values within the width set but will use more print space if required. By default, values are printed right aligned within a predefined display width.
dispMode promtAndEcho, displayResults ;	Sets the display mode. <u>promptAndEcho</u>: indicates whether the Justina prompt and user input echo must be displayed. Using predefined constants: NO_PROMPT 0 do not print prompt and do not echo user input PROMPT 1 print prompt but no not echo user input ECHO 2 print prompt and echo user input (default). <u>displayResults</u>: indicates if and how calculation results must be displayed. Using predefined constants: NO_RESULTS 0 do not print results RESULTS 1 print results (default) QUOTE_RES 2 print string values surrounded by double quotes. Backslash and double quote characters included in the string are expanded to escape sequences - see '*quote()*' function.

<u>Example</u>

```
Justina> 3 + 5
                                                 8
Justina> "abc\"def"
                                           abc"def
Justina> dispWidth 35
Justina> dispMode ECHO, QUOTE_RES
Justina> 3 + 5
                                  8
Justina> "abc\"def"
                          "abc\"def"
Justina>
```

Strings shown surrounded with double quote characters

With command `displayMode NO_PROMPT, RESULTS;` , Justina can be used as a ***programmable scientific calculator***, showing results not interrupted by prompts and user input echo.

Formatting numeric values

The two commands below define how floating-point and integer values are formatted when printed in calculation results and in echoed user input.

⚠ Note that these two settings also define how numeric values are printed using commands to write data to any input/output device or SD card file (if SD card connected). See chapter 10: *Input and output*.

floatFmt precision [, notation] [, flags]] ;	Sets display / print formatting for floating-point numbers. Arguments 'notation' and 'flags' can be entered as predefined constants. Precision: With fixed point and exponential notation, specifies the number of digits to be printed after the decimal point. With 'shortest' notation, specifies the maximum number of significant digits to be printed. Notation: display format for floating point numbers. FIXED "f" fixed point notation EXP "e" scientific notation EXP_U "E" scientific notation, 'E' uppercase SHORT "g" shortest notation (fixed or scientific) SHORT_U "G" shortest notation (fixed or scientific), 'E' uppercase Flags: used to finetune output. Flags are predefined constants. They can be combined by adding their values together. Using predefined constants: FMT_LEFT 1 align left FMT_SIGN 2 always add a sign (- or +) preceding the value FMT_SPACE 4 precede the value with a space if no sign FMT_POINT 8 always add decimal point FMT_000 16 pad the print field with zeros FMT_NONE 0 clear all flags (see remark, below)
intFmt precision [, notation] [, flags]] ;	Sets display / print formatting for integers. Arguments 'notation' and 'flags' can be entered as predefined constants. Precision: specifies the minimum number of digits to be written. If the value has less digits, the print field will be padded with leading zeros. Notation: DEC "d" decimal representation (base 10) HEX "x" hexadecimal representation (base 16) HEX_U "X" idem (base 16), hexadecimal digits A-F uppercase Flags: used to finetune output. Flags are predefined constants. They can be combined by adding their values together. FMT_LEFT 1 align left FMT_SIGN 2 always add a sign (- or +) preceding the value FMT_SPACE 4 precede the value with a space if no sign FMT_0X 8 if hex. notation: precede non-zero values with "0x" or "0X" FMT_NONE 0 clear all flags (see remark, below)

Notation and flag arguments are both optional; notation and flags last set remain in effect until explicitly entered as argument a next time the command is executed. When flags are included as argument, all flags not included are reset. To clear all flags explicitly, use value 0 (or use predefined flag FMT_NONE).

Note: to display **string** results left justified, set the display width to zero (or use the *fmt()* function - explained in chapter 10: *Input and output*).

Example

Display floating point number '12.3456789' using different settings.

```
Justina> floatFmt 10, FIXED
Justina> 12.34567928
                                  12.34567928

Justina>
Justina> floatFmt 10, FIXED
Justina> 12.34566975
                                  12.34566975

Justina> floatFmt 2, FIXED
Justina> 12.35
                                        12.35

Justina> floatFmt 2, EXP, FLAG_000
Justina> 1.23e+01
0000000000000000000000000000000000000001.23e+01
Justina>
```

The maximum number of digits after the decimal point is limited to 8; note the rounding error (accuracy floating point numbers)

Example

Display integer 53 padded with leading zeros; then display integer 1234567 using different settings.

Note that the number base used for input can be binary, decimal or hexadecimal, this is unrelated to the output format.

```
intFmt 8, DEC
Justina> 53
                                          00000053

Justina> intFmt 0
Justina> 123456
                                            123456

Justina> intFmt 8, HEX, FLAG_0X
Justina> 0x1e240
                                        0x0001e240

Justina> -0x1e240
                                        0xfffe1dc0

Justina>
```

Example

Perform binary arithmetic and use bitwise operators (discussed in chapter 7: *Operators*)

```
Justina> intFmt 0x8, HEX, FLAG_0X
Justina> 0x45a9 + 0x6be5
                                        0x0000b18e

Justina> (0x3039 & 0xd431) << 0x2
                                        0x000040c4

Justina>
```

6 Justina variables and constants

6.1 Variables

A variable can hold any of the three available data types: integer, float and string.

A variable is declared using the keyword **var**, the name of the variable and an optional initializer. A variable declared from the command line is a user variable; any variable declared within a program is a program variable.

- Within a program function, variables can also be declared with the 'static' keyword - see chapter 13: *Programming*.

Variables can be declared as **scalars** (holding one value) or **arrays** (holding multiple values).

- Scalar variables can receive values of any data type - they will adapt their value types accordingly.
- Arrays can have 1 to 3 dimensions. All values stored in an array have the same data type. Once initialized, **an array cannot change its data type** anymore. If possible, values will be cast to the data type of the array. Otherwise, an execution error will be produced.

var name1 [(dim1 [, dim2 [, dim3]]])] = literal1 [, name2 ..., name3 ...] ;

If a variable has an initializer literal, the data type is derived from it. Without an initializer, the variable is defined as a float and is initialized to zero. String arrays can only be initialized with an empty string.

The **var** command is a non-executable command: it creates and initializes variables before execution starts (during parsing).

Delete individual user variables with the **delete** command, followed by a list of variable names (arrays: without dimensions).

delete name1, name2, ... ;

The **delete** command is a non-executable command and it is not allowed within a program. It must be the first (or only) statement typed in the command line. It deletes user variables before execution starts.

This will produce an error:

```
var hello;
delete hello; hello = "hi";
```

error (variable does not exist)

Program variables are deleted automatically when a program is deleted.

Examples

```
var monthlyDetail (12, 5) = 0, monthlyTotal (12) = 0, grandTotal = 0;
var birdNames (10) = "";
delete monthlyDetail, birdNames;
```

The array initializers are important here because they declare the two arrays as integer arrays. By default, they would be declared as floats (and initialized to 0.)

The maximum for each dimension is 255 elements. But because of RAM memory constraints in microcontrollers, the maximum number of array elements is set to 1000, occupying a 4-kilobyte block of data. This does not include character storage for non-empty strings stored as array elements. For the same reason, string arrays are always initialized with empty strings.

Variable names

Variable names follow the same rules as names for constants and user function names: they must start with a letter from a to z (or A to Z) and may be followed by a sequence of letters, digits and underscore characters. The maximum name length is 20 by default. Names are case sensitive.

6.2 Constant variables

Just as normal variables, constant variables can hold any of the three available data types: integer, float and string.

A Justina constant variable is declared using the keyword '**const**' followed by the name of the constant, an equal sign and a constant literal defining a value in any of the three defined data types. A constant declared from the command line is a user constant; any constant declared within a program is a program constant.

Multiple constants can be declared in one '**const**' statement:

const name1 = literal1 [, name2 = literal2, name3 = literal3...] ;

The constant data type is derived from the initializer literal (which is mandatary for a constant). Once initialized (before execution starts), the contents of a constant cannot be changed any more.

The **const** command is a non-executable command: it creates and initializes constant variables before execution starts (during parsing).

Delete individual user (constant) variables with the **delete** command, followed by a list of user constant (and variable) names.

delete name1, name2, ...

Note: this is the same **delete** command used to delete non-constant variables. It's a non-executable command and it is not allowed within a program. It must be the first (or only) statement typed in the command line.

Note that program constants are only deleted when a program is deleted.

Examples

`const animal = "dog"`	Justina string
`const chairs = 3, height = 3.2, pet = "cat"`	3 constants defined
`delete tables, chairs;`	delete 2 constants

Constant names

Constant variable names follow the same rules as names for ordinary variables. Constant variables are always **scalar**, containing one single value; they cannot be defined as arrays of values.

6.3 Predefined constants

Constant variables are not to be confused with predefined constants, like:

e	refers to 'Eulers Number' (2.718281...)
PI	3.1415926535897932...
INPUT_PULLUP	is used as argument of Justina function '*pinMode()*'

During parsing, the symbolic name of a predefined constant is replaced by its value.

Using predefined constants as arguments to command or functions makes a program much more readable and understandable.

Boolean values

Justina uses the integer data type to work with Boolean values: a zero value means 'false', a non-zero value means 'true'. When Justina needs a Boolean value (e.g., as argument of a function), use a predefined constant instead of '0' or '1' to enhance for readability:

'Boolean' constant	Value
FALSE	0
TRUE	1
LOW	0
HIGH	1
OFF	0
ON	1

A complete list of predefined constants is available in Appendix H: *List of predefined constants.*

7 Operators

Justina operators are listed below with precedence (1 is highest) and associativity.

- Associativity: if two or more successive operators in an expression have **same precedence** (as in 1 + 2 - 5), then associativity defines whether operations will be applied to the operands in right-to-left or left-to-right order.
  Most operators have left-to-right associativity. Assignment operators (including compound assignment operators, like the '+=' operator), all prefix operators and exponentiation operator '**' have right-to-left associativity.
  As an example, the power operator '**' has right-to-left associativity:

  2 ** 3 ** 2 ⇔ 2 ** (3 ** 2) ⇔ 2 ** 9 = 512

Operators with right-to-left associativity are shown with a light gray background.

Precedence	Operator	Description
1 Highest	( )	Parentheses (function calls, array elements, simple parentheses)
2	++ --	Postfix increment and decrement
3	++ --	Prefix increment and decrement
4	**	power
5	+ - ! ~	Unary plus and minus Logical negation Bitwise complement
6	* / %	Multiplication, division Integer modulus
7	+ -	Addition, string concatenation Subtraction
8	<< >>	Bitwise shift left and right
9	< <= > >=	Relational less than, less than or equal Relational greater than, greater than or equal
10	== !=	Relational is equal, is not equal
11	&	Bitwise and
12	^	Bitwise exclusive or
13	|	Bitwise inclusive or
14	&&	Logical and
15	||	Logical or

16	= *= /= += -= %= &= ^= |= <<= >>=	Assignment ***Compound assignments*** Multiplication / division assignment Addition, string concatenation assignment Subtraction assignment Modulus assignment Bitwise and, exclusive or, inclusive or assignment Bitwise shift left and right assignment
17 Lowest	,	Separator between expressions (arguments) within a command or function
18 Lowest	;	Separator between statements

The precedence and associativity rules are almost identical to those in c++. If you are in doubt, or to enhance readability: use parentheses!

<u>Notes</u>

in Justina, an expression containing an assignment is still an expression.

So, the following expressions are perfectly valid:

```
a = b = c;          ⇔  a = ( b = c );          ⇔  b = c; a = b;
a = 1 + (b+= c);    ⇔  a = 1 + (b = b + c);    ⇔  b = b + c; a = 1 + b;
```

All operators in the table above require numeric values as operands. Exception: the addition operators (=   +=) are used as string concatenation operators when the operands are strings.

Compound assignments first perform an operation on the two operands and then assign the result to the first operand. Example: a+=2 adds two to the value of a (which must be a variable).

Bitwise operators (& | ^   ~   <<   >>   &= |=   ^=   <<=   >>=) and the integer modulus operator (%) need **integer values** as operands. Applying these operators to floating point operands will create a runtime error (execution error).

Note: to calculate the modulus of two floating point numbers, use the modulus function *fmod()* described in chapter 8: *Math, string, type conversion, test and lookup functions*.

8 Math, string, type conversion, test and lookup functions

The Justina interpreter comes with many built-in ('internal') functions. This chapter covers part of them. Arduino specific functions will be discussed in the next chapter. Other functions are covered in specific chapters, e.g., input and output functions.

☞ Note that Justina function names, like other identifiers in Justina, are case sensitive (variables, symbolic constants, command names and function names whether they are built-in or written by yourself as in the last example).

In the remainder of this chapter, both the term 'value' and 'expression' refer to an expression that will be evaluated to obtain a value, unless otherwise noted.

8.1 Math functions

angleMode mode ;	This is not a function but a **command**. It sets the angle mode for trigonometric functions. Using predefined constants: RADIANS 0 set angle mode radians DEGREES 1 set angle mode degrees

Functions in this table always return a float. The argument(s) must be integer or floating-point numbers.

sqrt (value)	square root
sin (value)	sine of an angle. Angle: radians or degrees (see angle setting)
cos (value)	cosine of an angle. Angle: radians or degrees (see angle setting)
tan (value)	tangent of an angle. Angle: radians or degrees (see angle setting)
asin (value)	inverse sine. Returns an angle in radians or degrees (see angle setting)
acos (value)	inverse cosine. Returns an angle in radians or degrees (see angle setting)
atan (value)	inverse tangent. Returns an angle in radians or degrees (see angle setting)
ln (value)	natural logarithm
lnp1 (value)	*ln* (value+1). Tends to be more accurate than *ln(*value+1) for small values
log10 (value)	common (base 10) logarithm
exp (value)	natural exponential. Same as e ** value
expm1 (value)	(e ** value) - 1. Tends to be more accurate than (e ** value) for small values
round (value)	rounds value to the closest integer
ceil (value)	rounds value to the closest integer not less than value
floor (value)	rounds value to the closest integer not greater than value
trunc (value)	rounds value towards zero. Example: '*trunc*(-5.7)' yields -5.
fmod (value)	remainder of division (note: for integer division remainder, use *operator* '%'

The function below always returns an integer, even if the argument is a floating-point number.

The argument must be integer or floating-point numbers.

signBit (value)	sign bit of numeric value (integer or floating-point number): the sign bit of negative numbers is 1; for zero and positive numbers it will be 0.

Functions in this table return a float if the argument / at least one of the arguments is a floating-point number, otherwise an integer is returned. The argument(s) must be integer or floating-point numbers.

min (value1, value2)	minimum of two values
max (value1, value2)	maximum of two values
abs (value)	absolute value

A number of mathematical constants are predefined in Justina (e, π, ...), as well as conversion factors from radians to degrees and vice versa. Please refer to Appendix H: *List of predefined constants*.

8.2 Lookup and test functions

ifte (test value, value if true, value if false) *ifte* (test value 1, value if true, test value 2, value if true [, test value n, value if true ...] [value if false])

The first form corresponds to the classic if (...) function. The test expression is evaluated, if it is true (not equal to zero) the 'value if true' is returned, otherwise the 'value if false' is returned.

The second form successively evaluates test values from left to right until a test result is true (not equal to zero). It then returns the corresponding 'value if true'. If none of the test values evaluate to true, either a zero is returned or, if provided, the 'value if false' is returned.

Test values must be numeric; other arguments can be any data type.

Note that **all** arguments of these functions are evaluated.

Maximum number of function arguments = 16.

switch (value, test 1, result 1 [, test 2, result 2 ...] [default result])

The first argument ('value') is successively compared with the test values 'test 1', 'test 2', ... and if a match is found, the corresponding result value is returned. If no match is found, either zero is returned or the default result (if it is provided).

All data types are accepted as function arguments.

Note that **all** arguments of this function are evaluated.

Maximum number of function arguments = 16.

choose (index value, value if 1, value if 2 [value if 3, value if 4, ...])

The index value is an integer not smaller than one. It determines which of the next values will be returned.

An error is produced if the index is not within range (1 to the number of return values provided).

Except for the index value, all data types are accepted as arguments.

Note that **all** arguments of this function are evaluated.

Maximum number of function arguments = 16.

index (test expression, expression 1, expression 2 [, expression 3 ...])

The test expression is successively compared with the other expressions provided as arguments until a match is found and the index number of the match is returned. If no match is found, zero is returned.

All data types are accepted as function arguments. The data type of the function result is integer.

Note that **all** arguments of this function are evaluated.

Maximum of function arguments = 16.

8.3 Type conversion functions

cInt (value)	Attempts to convert a value in any data type to an integer value. if the argument is a string, characters will be taken into account as long as the resulting value is a valid integer. If none, zero will be returned.
cFloat (value)	Attempts to convert a value in any data type to a floating-point value. if the argument is a string, characters will be taken into account as long as the resulting value is a valid floating-point number. If none, zero will be returned.
cStr (value)	Converts a value in any data type to a string. No specific formatting will be applied (to **format** a value into a string, use the *fmt*(...) function.

8.4 String functions

Functions within these tables all deal with strings (a sequence of characters).

Functions referring to a position within a string use 1 as the first character. The position of the last character indicates the length of a string. An empty string has 0 characters.

If character positions or other arguments are outside the valid range, an error will be produced.

char (asciiCode)	Returns a one-character string with the character represented by asciiCode (0 <= asciiCode <= 0xFE). 0xFF is not considered a valid ASCII code.
len (string)	Returns string length
line ()	Returns a 2-character string with a Carriage Return Line Feed Sequence
asc (string [, charPos])	Returns the ascii code for string character indicated by charPos (1 to n). Default charPos = 1
rtrim (string)	Returns a string with trailing spaces removed
ltrim (string)	Returns a string with leading spaces removed
trim (string)	Returns a string with leading and trailing spaces removed
left (string, n)	Returns a string containing only the n leftmost characters
mid (string, start, n)	Returns a string containing the n characters starting at position start of the original string
right (string, n)	Returns a string containing only the n rightmost characters
toUpper (string [, start [, end]])	Returns a string with (part of) the original string converted to uppercase
toLower (string [, start [, end]])	Returns a string with (part of) the original string converted to lowercase
space (n)	Returns a string containing n spaces
repeatChar (string, n)	Returns a string with character 1 from a string repeated n times
findStr (string, substring, [, start])	Returns the position of a substring in a string. If a start position for the search is not given, the search starts at the first character. Returns 0 if substring not found

replaceStr (string, substring, replaceWith [, start])	Returns a string with a given substring substituted by a replacement string. If a start position for the search is not given, the search starts at the first character. Returns the original string if substring is not found. If start is a variable, its value will be set to the first position in the returned string after the replacement string. Returns 0 if the substring was not found, 1 + new string length if substring contained last characters of string
replaceChar (string variable, ASCII code [, character position])	Replaces a single character in a string variable with a character specified by its ASCII code. If 'character position' is not specified, the first character is replaced. As this changes the original string object and no new string is created, this speeds up execution. This function always returns zero.
strCmp (string1, string2)	Performs a binary comparison between two strings. Returns 0 if the two strings are equal. Returns a negative integer if the first non-matching character has a lower value in string 1 than in string2 and a positive integer if it has a greater value
strCaseCmp (string1, string2)	Performs a case **insensitive** comparison between two strings. Returns 0 if the two strings are equal. Returns a negative integer if the first non-matching character has a lower value in string 1 than in string2 and a positive integer if it has a greater value
ascToHexStr (ASCII code)	Returns a two-character string encoding a given ASCII code into two characters, representing the two hexadecimal digits of the ASCII code. Examples: *ascToHexStr*(0x61) -> "61" (ASCII code of 'a') *ascToHexStr*(98) -> "62" (ASCII code of 'b') *ascToHexStr*(*asc*("c")) -> "63" (ASCII code of 'c')
hexStrToAsc(string [, start position])	Decodes two characters of a string, starting at 'character position' and representing the two hexadecimal digits of an ASCII code, into that ASCII code. If 'start position' is omitted, the first two characters are decoded. If the characters do not represent hexadecimal digits, the function returns -1. Examples: *hexStrToAsc* ("62") -> 98 (ASCII code of 'b') *char*(*hexStrToAsc* ("63")) -> "c"
quote (expression)	If the argument is a number, converts it to a string (same as *cStr()* function). If the argument is a string: • add surrounding double quotes • replace all '\' (backslash) characters in the string with a sequence of two '\' characters • replace all ' " ' (double quote) characters with a sequence consisting of a '\' character and a ' " ' character Note: *quote()* is most useful when used with the *eval()* function (further down in this chapter).

In the following table, the default for argument 'charPos' is 1.

isAlpha (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a letter
isAlphaNumeric (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a letter or a digit
isAscii (string [, charPos])	Returns a non-zero value if the character indicated by charPos is an ASCII character
isControl (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a control character (ASCII codes 0 to 0x1f; 0x7f)
isDigit (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a digit
isGraph (string [, charPos])	Returns a non-zero value if the character indicated by charPos has a graphical representation. Same as *isPrintable()* function, but without the space character
isHexDigit (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a hexadecimal digit (0 to 9, A to F)
isLowerCase (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a lowercase character (a to z)
isUpperCase (string [, charPos])	Returns a non-zero value if the character indicated by charPos is an uppercase character (A to Z)
isPrintable (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a printable character (it's not a control character): all ASCII codes greater than 0x1f, except ASCII code 0x7f
isPunct (string, [, charPos])	Returns a non-zero value if the character indicated by charPos has a graphical representation (as in *isGraph()* function) but is not alphanumeric
isWhitespace (string [, charPos])	Returns a non-zero value if the character indicated by charPos is a space, a horizontal tab (0x09), a vertical tab (0x0b), a form feed (0x0c), a carriage return (0x0d) or a new line (0x0a) character

8.5 Information functions

ubound (array variable name, dimension)	Array variables only: returns the upper bound of a dimension (an array can be defined with 1 to 3 dimensions). If the variable is not an array, or the dimension specified does not exist, an error is returned
dims(array variable name)	Array variables only: returns the number of dimensions (an array can be defined with 1 to 3 dimensions). If the variable is not an array, or the dimension specified does not exist, an error is returned
type (expression)	Returns the variable (scalar or array element) data type. Use following constants to test the data type of a value: INTEGER 1 integer data type

<table>
<tr><td></td><td>FLOAT 2 floating point data type
STRING 3 string data type
Note: although all elements of an array have the same data type, you must specify an array element</td></tr>
<tr><td>r ([index])</td><td>If 'index' is 1 or index is not provided, returns the last result of a calculation.
If 'index' is between 2 to 10, return previous results.</td></tr>
<tr><td>isColdStart()</td><td>Returns 1 if Justina went through a cold start; returns 0 if not (please refer to the quit command)</td></tr>
<tr><td>sysVal (index)</td><td>Returns a system value maintained by Justina.
<table>
<tr><td>Index</td><td>Description</td></tr>
<tr><td colspan="2">Display settings</td></tr>
<tr><td>0</td><td>display width (calculation results only)</td></tr>
<tr><td>1</td><td>floating point number formatting: precision</td></tr>
<tr><td>2</td><td>floating point number formatting flags</td></tr>
<tr><td>3</td><td>floating point number formatting: notation</td></tr>
<tr><td>4</td><td>integer number formatting: precision</td></tr>
<tr><td>5</td><td>integer number formatting flags</td></tr>
<tr><td>6</td><td>integer numbers formatting notation</td></tr>
<tr><td>7</td><td>prompt and echo setting (none, prompt only, prompt + echo)</td></tr>
<tr><td>8</td><td>print last calculation results: yes/no</td></tr>
<tr><td>9</td><td>angle mode: 0 is radians, 1 is degrees</td></tr>
<tr><td colspan="2">fmt() function settings</td></tr>
<tr><td>10</td><td>print field width</td></tr>
<tr><td>11</td><td>numeric values: precision</td></tr>
<tr><td>12</td><td>numeric values: formatting flags</td></tr>
<tr><td>13</td><td>numeric values: notation</td></tr>
<tr><td>14</td><td>string values: number of characters to print</td></tr>
<tr><td colspan="2">other</td></tr>
<tr><td>15</td><td>Current count of ‘last values’ stored in ‘last values fifo’</td></tr>
<tr><td>16</td><td>open SD file count</td></tr>
<tr><td>17</td><td>number of defined external IO devices</td></tr>
<tr><td>18</td><td>loaded program name (or empty string)</td></tr>
<tr><td>19</td><td>trace string (if defined, otherwise empty string)</td></tr>
<tr><td colspan="2">product info</td></tr>
<tr><td>31</td><td>product name</td></tr>
<tr><td>32</td><td>legal copyright</td></tr>
<tr><td>33</td><td>product version</td></tr>
<tr><td>34</td><td>build date</td></tr>
<tr><td colspan="2">technical data (normally not relevant for the user)</td></tr>
<tr><td>36</td><td>evaluation stack: element count</td></tr>
<tr><td>37</td><td>flow control stack: element count (call stack depth + open block count)</td></tr>
<tr><td>38</td><td>call stack depth</td></tr>
<tr><td>39</td><td>number of stopped programs</td></tr>
<tr><td>40</td><td>parsed programs stack: element count (number of stopped programs + open eval() strings</td></tr>
<tr><td>41</td><td>created linked list object count (since startup)</td></tr>
</table>
</td></tr>
</table>

	42	currently active (created, not yet deleted) objects: count per object type
	43	currently active (created, not yet deleted) objects: errors per object type
	44	returns processor board type: BOARD_OTHER 0 none of the following BOARD_SAMD 1 SAMD arch.: nano 33 IoT,... BOARD_RP2040 2 nano RP2040 BOARD_ESP32 3 nano ESP32

8.6 The 'eval()' function: parsing and executing expressions at runtime

When an *eval()* function is executed, it stops execution, then parses a list of expressions stored in its string argument (using the same parser that parses Justina statements) and executes them. When done, normal execution continues.

eval (string)	String: a list of expressions, stored as text and separated by semicolons, and contained within double quotes. The string cannot contain command statements (expressions only) but no other restrictions apply: you may use variables and constants, operators, call built-in functions, functions in a Justina program, 'external' functions you write in c++, and even other (nested) *eval* functions. Function *eval()* returns the result of the last expression in its expression list as function result.

To include a string constant within 'string', use escape sequences (see chapter 4: *Data types*), like in this example (extra spaces for clarity): `eval("  \"abc\" + \"def\"  ");` returns string "abcdef".

Uses of the eval() function

1. Store much-used expressions as a string in a variable. You can then perform '*eval*(variable)' to obtain the result without having to write a Justina function.
2. Parse and evaluate an expression only known at runtime. Typical use: in conjunction with the **input** statement (see chapter 10: *Input and output*), allow the user to type in an expression (not just a value) when a program requests input from the user.
3. During debugging, access local variables of a program stopped for debugging, from within another program. Please refer to chapter 13: *Programming* for an example.
4. Use *eval()* as a (very simple) form of indirection (although in most cases, a better way is using an array).

Example: store much-used expressions as a string in a variable

In this example, a simple formula 'age * 12' is stored in variable ' yearsToMonths '.

Executing ' *eval*(yearsToMonths) 'returns the value stored in variable age, multiplied by 12.

```
Justina> var age, yearsToMonths = "age * 12"
Justina> age = 17; eval(yearsToMonths)
                                              204

Justina>
```

Example: evaluate an expression only known at runtime

One of the programs in the Justina library 'Examples' collection is stored in file ‘input.jus’. We will not study this program here; we'll merely use it to demonstrate the use of the *'eval()'* function.

Starting the program (function *evalInput()*), this is what you'll see:

```
Justina> evalInput()

===== Input (\c to cancel): =====
Please specify amount in metric ton
```

Apparently, the program has stopped, asking you to enter an amount in metric tons (the '**input**' statement taking care of this will be discussed in chapter 13: *Programming*).

If you enter `2 + 5 + 1;` and press ENTER, then you'll see this:

```
Justina> evalInput()

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 8000 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton
```

Within the program, an '*eval()*' function (see code line below) parses and executes expression '2 + 5 + 1', yielding 8, which is then multiplies by 1000 and stored in a program variable 'amount'. This amount is then added to variable 'totalAmount' (assignment operators have right-to-left associativity).

```
...
totalAmount += amount = eval(answer) * 1000;
...
```

Then, the program continues, first printing the amount in kg, and then asking for a new amount.

After a few entries, we exit the loop by typing '\c' (cancel) + ENTER.

The program then prints the **total** amount entered and exits.

```
Justina> evalInput()

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 8000 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 14000 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 6000 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton

*** total amount = 28000 kg

                                                  0

Justina>
```

Example: use eval() as a way to obtain indirection

This example uses two variables:

- variable 'ref' contains the name of another variable, in this example named 'value'
- variable 'value' contains value 7

```
Justina> var value = 7
Justina> var ref = "value"
Justina> eval(ref)
                                                7

Justina>
```

Executing 'ref' would return string "value", whereas *'eval*(ref)' returns numeric value 7.

Please note that this is not a very elegant way of 'calculating' which value you want to obtain ('7').
A much better method is storing values in an array and then simply indexing the array.

9 Arduino-specific functions

The functions below are, in most cases, the Justina equivalent of corresponding Arduino functions. Use them in your Justina programs or type them in from the command line of the Serial Monitor for quick prototyping or testing.

9.1 Arduino-specific digital I/O, timing and other functions

To refer to built-in LED pins, use these predefined constants:

Arduino constant	Value	
LED_BUILTIN	13	
LED_RED	14	(Arduino nano ESP32 only)
LED_GREEN	15	(Arduino nano ESP32 only)
LED_BLUE	16	(Arduino nano ESP32 only)

The following Justina functions implement the corresponding Arduino functions. Please visit the *Arduino Language Reference* for accurate descriptions.

millis ()	
micros ()	
delay (time in milliseconds)	In Justina, ***this function is replaced by the wait () function***, described below this table.
digitalRead (pin)	returns 0 (pin value is low) or 1 (high)
digitalWrite (pin, value)	value: for readability, use these predefined constants: LOW or OFF or FALSE 0 HIGH or ON or TRUE 1
pinMode (pin, mode)	mode: for readability, use predefined Arduino constants INPUT 1 OUTPUT 3 INPUT_PULLUP 5 INPUT_PULLDOWN 9
analogRead (pin)	
analogReference (type)	Not for nano RP2040
analogWrite (pin, value)	
analogReadResolution (bits)	
analogWriteResolution (bits)	
noTone (pin)	
pulseIn (pin, value [, timeout])	value: use predefined constants LOW (0), HIGH (1)
shiftIn (dataPin, clockPin, bitOrder)	bitOrder: for readability, use predefined constants LSBFIRST 0 MSBFIRST 1
shiftOut (dataPin, clockPin, bitOrder, value)	bitOrder: use predefined constants LSBFIRST, MSBFIRST

tone (pin, frequency [, duration])	
random ([min,] max)	
randomSeed (seed)	

9.2 Justina functions replacing Arduino-specific functions

wait (time in milliseconds)	This is the Justina replacement of the Arduino delay(..) function. It does exactly the same thing (waiting for a number of milliseconds), but **without suspending** Justina background tasks (e.g. maintaining a TCP connection) while waiting. More information is available in *Appendix D: 'Running background tasks: system callbacks'.*

9.3 Arduino-specific bit and byte manipulation functions

The following functions implement the corresponding Arduino functions. Please visit the *Arduino Language Reference* for accurate descriptions.

x: value to be read or changed, n: bit number (0 to 31); b: bit value to write (0 or 1).

x, n data type must be integers.

Note that in the standard Arduino functions, the value to be changed, x, must be a variable. In Justina, a constant is allowed too (constants are not modified, of course).

bit (n)	
bitRead (x, n)	
bitClear (x, n)	***Justina addition****:* x can also be an integer constant
bitSet (x, n)	***Justina addition****:* x can also be an integer constant
bitWrite (x, n, b)	***Justina addition****:* x can also be an integer constant
highByte (x)	***replaced by Justina function byteRead (see below)***
lowByte (x)	***replaced by Justina function byteRead (see below)***

9.4 Additional Justina bit and byte manipulation functions

These functions provide some useful *additions* to the Arduino-specific functions implemented in Justina.

In the tables underneath, x represents the value to be read or changed. Data type: integer.

Byte read and write functions

Reads or writes 1 byte of a 32-bit integer value (constant or variable).

Note: use *byteRead()* instead of Arduino lowByte and highByte functions, which are not supported.

x: value. Data type: integer

n: byte number (from 0 to 3; 0 is low order byte). Data type: integer.

b: value to write (0 to 255). Data type: integer.

byteRead (x, n)	returns byte n of an integer value (number between 0 and 255)
byteWrite (x, n, b)	returns x, with byte n of x changed to the lowest 8 bits of b. Other bits of x are unchanged. If x is a variable, its value will be set to the returned function result as well. This can be avoided by putting x between parentheses Example: *fmt*(byteWrite (0xF0F0, 1, 0x66), "x") -> 0x66F0

Masked word read and write functions

Reads a masked 32-bit value; writes, sets or clears bits in a 32-bit value, specified by mask.

All arguments must have an integer data type.

maskedWordRead (x, mask)	Returns value x with mask applied
maskedWordClear (x, mask)	Returns value x with the bits indicated by mask cleared. If x is a variable, its value will be set to the returned function result as well. This can be avoided by putting x between parentheses
maskedWordSet (x, mask)	Returns value x with the bits indicated by mask set. If x is a variable, its value will be set to the returned function result as well. This can be avoided by putting x between parentheses
maskedWordWrite (x, mask, v)	Returns x with bits indicated by mask changed to same bits in v. If x is a variable, its value will be set to the returned function result as well. This can be avoided by putting x between parentheses

9.5 Direct memory location read and write functions

Useful to read specific memory locations, for instance peripheral registers (input / output, timers, ...), if you have good reasons not to use the Arduino functions provided or if there is no Arduino function available.

WARNING

Only use these functions if you really know what you're doing.
If not, disaster will be lurking around the corner.

a: memory address as a 32-bit integer value (e.g., 0xa0f52804). The functions below will align the address with the start of a 32-bit word before executing the function.

n: byte number in a word (0 to 3; 0 is low order byte).

v: value to read or write.

All arguments must have an integer data type.

mem32Read (a)	returns the 32-bit word stored at memory address a as an integer
mem32Write (a, v)	writes a 32-bit integer value v to memory address a. The function returns 0
mem8Read (a, n)	returns the 8 bits stored at memory address a, byte n as an integer
mem8Write (a, n, v)	writes an 8-bit integer value v to memory address a, byte n. The function returns 0

10 Input and output

10.1 Introduction

By default, Justina uses the Arduino Serial monitor (or any serial terminal program or device) as its only IO device. However, when the Arduino program creates the Justina object, it can pass a reference to all 'external IO' stream objects it wants to make available as IO devices to Justina. These can be Serial ports, a TCP IP client, an LCD or OLED display... (for more information, see *Appendix A: Creating a Justina object and choosing* startup options). The maximum number of IO devices that can be defined in Justina is 4.

Note: typing '*sysVal*(17)' will return the number of IO devices defined.

Justina handles input and output from/to I/O devices and (if an SD card board is connected) SD card files in the same way, using a set of common commands and functions (SD card commands and functions which are not applicable to IO devices will be discussed in next chapter: *Working with SD cards)*.

- IO devices are referred to by an assigned 'device number': a negative number *from minus 1 to minus 'the number of IO devices'* in the order the stream references are passed to Justina.
- SD card open files are referred to by an assigned 'open file number': a positive number from 1 to the number of currently open SD card open files.

Generically, IO devices and open files are referred to as streams and are referred to by stream numbers.

Various Justina functions and commands require a device number or open SD file number as argument.

Predefined constants are available to represent IO devices and open files in Justina (use them to make a program more readable):

constant	IO device number	constant	open file number
IO1	-1	FILE1	1
IO2	-2	FILE2	2
IO3	-3	FILE3	3
IO4	-4	FILE4	4
CONSOLE	0 (see below)	FILE5	5

⚠ You should also read next chapter: *Working with SD cards, if you plan to use SD card functionality*

The console

The console is defined as the only IO device capable of sending Justina commands, typed in the command line of the Arduino IDE Serial monitor (or a suitable Terminal application on your PC or even on your smartphone) to Justina. It is also the IO device where system messages, the echo of statements, results of calculations, ... and the *'Justina>'* prompt are sent to.

At startup, the IO device referred to by device number -1 (IO1) is set as the console. The user can change the console to another IO device (if available).

You can read from, and write to, the console without having to bother about its device number in 2 ways:

- several functions and commands always read from / write to the console. Example: *cin()* function, **cout** commands.
- use predefined constant CONSOLE as device number with commands and functions requiring a device number.

Debug out

During debugging and tracing, Justina writes specific information (e.g., the source line where a program was stopped) to either a designated IO device or a designated SD file (if an SD card board is connected). This IO device or file is simply named 'debug out' (files will be discussed in next chapter: *Working with SD cards*, debugging and tracing in chapter 13: *Programming*).

In addition, you can write to 'debug out' with two specific commands, **dbout** and **dboutLine**.

The debug out device (or file) is especially useful while debugging a Justina program.

At startup, IO device -1 (IO1) is set as the debug out device: because IO1 is also the (default) console, messages sent in the context of debugging and tracing will appear on the console in between other system messages, your program output etc.

If this is not wanted, 'debug out' can be set to a different IO device or even an *open SD card file* (if an SD card board is connected).

10.2 Printing data to a stream

The commands in the tables below will print all arguments, one by one, to the designated output device (or open SD card file).

- Two commands, **vprint** and **vprintLine**, do not print to an IO device or open file but to a variable.

Functions *fmt(), tab(), col()* and *pos()* can be used to format individual arguments (see section 1.2.10: *Applying formatting to your output)*.

If no formatting is applied, floats and integers will be printed according to their respective display settings (see **floatFmt** *,* **intFmt** commands in chapter 5: *The console*), but without taking into account any formatting flags set there:

- integers printed in hex format will be preceded by '0x'
- floats will always print with decimal point
- strings will print without any truncating

Printing to console

Note: any *external* IO device can be set as console (see further).

cout arg1 [, arg2, arg3, ...] ;	Print all arguments to the console
coutLine [arg1 , arg2, arg3, ...] ;	Same as **cout**, but advance to a new line when done

Example

In the command line, type

```
coutLine "an integer: ", 3 * 5, line(), "a float: ", 3. * 5.;
```

Serial monitor output:

```
Justina> coutLine "an integer: ", 3 * 5, line(), "a float: ", 3.00 * 5.00
an integer: 15
a float: 15.00
Justina>
```

The '*line ()*' function (third argument) advances the print position to the start of a new line. When all arguments are printed, the print position moves to the next line (**coutLine** command) and the prompt is printed.

In this case, using **cout** instead of **coutLine** would have had the same effect, because before printing its prompt, Justina **always** goes to a new line.

Print to debug out

The syntax of these commands is identical to the syntax of the console print commands.

Note: any 'stream' (external IO device or open SD file) can be set to 'debug out'.

dbout arg1 [, arg2, arg3, ...] ;	Print all arguments to debug out
dboutLine [arg1 , arg2, arg3, ...] ;	Same as **dbout**, but advance to a new line when done

Print to any output device

These commands take one additional argument: a 'stream' number. Negative 'stream' numbers (or constants IO1 to IO4) refer to an external IO device (Serial port, TCP IP client, LCD screen...), positive numbers to an open SD file.

Apart from the 'stream' number (first argument) the syntax of these commands is identical to the syntax of the console print commands.

print streamNumber, arg1 [, arg2, arg3, ...] ;	Print all arguments to a stream
printLine streamNumber [, arg1, arg2, arg3, ...] ;	Same as **print** command, but advance to new line when done

Example

In the command line, type

```
printLine IO1, "name ", col(10), "John", line(), " 2 4 6 8 0";
```

Serial monitor output:

```
Justina> printLine IO1, "name ", col(10), "John", line(), " 2 4 6 8 0"
name     John
 2 4 6 8 0
Justina>
```

The predefined constant IO1 refers to IO device -1, which is set as the console, so print output is sent to the console (we could also have used predefined constant CONSOLE, or simply '-1' or '0').

The *col()* function moves the print column to column 10 before printing "John" (see further).

Print to a variable

These commands do not print to a stream but to a variable.

This allows you to create a string containing formatted data without actually printing it.

The variable must be able to accept 'string' as data type: if an array element, the array should be defined as string array (arrays cannot change their data type).

vprint variable, arg1 [, arg2, arg3, ...] ;	Print all arguments to a variable
vprintLine variable [, arg1, arg2, arg3, ...] ;	Same as **vprint**, but add CR and LF characters

Example

In the command line, type these 3 commands (variable 'test' should not exist yet):

```
var test = "before";
vprint test, "after:", tab(), "Pi =", PI;
cout test;
```

Serial monitor output (assuming that fixed point notation with 2 digits after the decimal point is set for floating point numbers):

```
var test = "before"
Justina> vprint test, "after: ", tab(), "PI = ", 3.14
Justina> cout test
after:  PI = 3.14
Justina>
```

The *tab()* function moves the print position to the start of the next group of print columns. See further down in this chapter.

Printing comma-separated argument lists to a stream or variable

These commands print a *comma separated* list that can later be parsed again into separate variables (with functions *cinList()*, *readList()* and *vreadList()*).

- floats will be printed with all significant digits, integers in decimal format (base 10)
- strings will be printed *with surrounding quotes*.
  - backslash (\ ) characters found will be replaced by a sequence of two backslash characters (\\ ) (spaces added here for clarity)
  - double quote (") characters found will be replaced by a sequence of a backslash and double quote character (\") (spaces added here for clarity)

At the end, an end of line sequence is added (CR and LF characters).

When printing comma-separated argument lists, display settings for integers and floats (**intFmt**, **floatFmt** commands) are not considered.

Although the primary use of these commands is writing data to SD files in a format that allows to easily retrieve it later (SD card and files will be treated in a separate chapter), these commands write to any valid stream.

coutList arg1 [, arg2, arg3, ...] ;	prints a comma separated list to the console.
printList streamNumber, arg1 [, arg2, arg3, ...] ;	Same as **coutList**, but prints to any stream (external IO or open SD file).
vprintList listVariable, arg1 [, arg2, arg3, ...] ;	Same as **coutList**, but prints to a variable.

Examples

In the command line, type these 3 commands:

```
var n1= 123, t1 = "abcdef", n2=456.789e10;
coutLine line(), n1, t1, n2, line();
outList n1, t1, n2;
```

Serial monitor output:

```
Justina> var n1 = 123, t1 = "abcdef", n2 = 4567890132992.00
Justina> coutLine line(), n1, t1, n2, line()

123abcdef4567890132992.00

Justina> coutList n1, t1, n2
123, "abcdef", 4.56789E+12
Justina>
```

Extra *'line()'* arguments have been included to improve clarity in this example.

Now, let's include a backslash and a double quote in variable t1.

```
t1 = "ab\\cd\"ef";
coutLine line(), n1, t1, n2, line();
coutList n1, t1, n2;
```

Remember, when entering text, precede a quote with a backslash and enter a backslash as a sequence of two backslash characters.

Serial monitor output:

```
Justina> t1 = "ab\\cd\"ef"
                                                  ab\cd"ef
Justina> coutLine line(), n1, t1, n2, line()

123ab\cd"ef4567890132992.00

Justina> coutList n1, t1, n2
123, "ab\\cd\"ef", 4.56789E+12
Justina>
```

1.2.10 Applying formatting to your output

The various print commands, described during the previous chapter, output data using default formatting.

The *fmt()* function is used to format the data the way you want before printing. It is most useful when it is used as an argument of a print command.

The meaning of arguments 'field width', 'precision', 'notation' and 'flags' corresponds to the definition of the same arguments used in the c++ printf function.

The function result is always a string, containing the formatted value.

fmt (expression [, field width [, precision [, notation] [, flags [, character count]]]] *fmt*(expression   [, precision] , notation [, flags [, character count]]

expression: the value to be formatted (numeric or string)

field width: the minimum print field width (formatted values will never be truncated). If less space is needed, the output will be padded with spaces.

precision: a) 'expression' evaluates to a string: the *maximum* number of characters to print if the string is longer.

b) 'expression’ evaluates to a number and 'notation' (see below) is DEC, HEX or HEX_U: if the value to be formatted is a float point number, it is first truncated (rounded towards zero) to an integer. 'precision' specifies the *minimum* number of digits of the integer value to be written. If the value has less digits, it will be padded with leading zeros.

c) 'expression' evaluates to a number and 'notation' (see below) is FIXED, EXP, EXP_U, SHORT or SHORT_U: if the value to be formatted is an integer, it is converted first to a float. 'precision' specifies the number of digits to be printed after the decimal point (fixed point, exponential notation) or the *maximum* number of significant digits to be printed (shortest notation).

notation:	a) 'expression' evaluates to a number: specifies how numbers should be represented.		

- As a floating-point number In fixed point notation, scientific notation or in the shortest
- As an integer: in decimal or hexadecimal representation.

Use the following constants to set a notation (same constants used in commands **intFmt** and **FloatFmt**):

FIXED	"f"	fixed point notation
EXP	"e"	scientific notation
EXP_U	"E"	scientific notation, ‘E’ uppercase
SHORT	"g"	shortest notation (fixed or scientific)
SHORT_U	"G"	shortest notation (fixed or scientific), ‘E’ uppercase
DEC	"d"	decimal representation (base 10)
HEX	"x"	hexadecimal representation (base 16)
HEX_U	"X"	idem (base 16), hexadecimal digits A-F uppercase

b) 'expression' evaluates to a string:

CHARS	"s"	character string

flags: Use the following flags to finetune the output (same constants used in commands **intFmt** and **FloatFmt**):

FMT_LEFT	1	align output left within the print field
FMT_SIGN	2	always add a sign (- or +) preceding numeric values
FMT_SPACE	4	precede numeric values with a space if no sign is written
FMT_POINT	8	print floating point numbers: always add decimal point
FMT_0X	8	print in hexadecimal notation: precede non-zero values with '0x' or '0X'
FMT_000	16	print floating point numbers: pad print field with zeros.
FMT_NONE	0	clear all flags

Character count: variable, will be updated with length of the formatted string returned. Combined with the *pos()* function, the current print column can be calculated. This can be useful to allow overlapping of print fields (see *pos()* and *col()* functions). An example is included in the Justina library.

All arguments following the value to be printed are ***optional*** (please see '*fmt*' function syntax to check out the allowed combinations of arguments).

The width, precision, notation and flags arguments remain in effect during next *fmt()* calls *until explicitly changed by next calls* to this function. When flags are included as argument, all flags not included are reset. To clear all flags explicitly, use value 0 (or use predefined flag FMT_NONE).

Examples

The following examples direct their output to the console. The "==" fields are printed to indicate the start and end of the formatted print field.

This first example (below) prints numbers in decimal and hexadecimal notation. The print width is set to 8 characters. Hexadecimal numbers are printed with at least 4 digits, the last number (a float) is first truncated to an integer and is printed left aligned.

```
Justina> cout "==", fmt(12, 8, 1, DEC), "=="
==      12==
Justina> cout "==", fmt(1234, HEX), "=="
==     4d2==
Justina> cout "==", fmt(1234, 4, HEX), "=="
==    04d2==
Justina> cout "==", fmt(1234, HEX, FLAG_0X), "=="
==  0x04d2==
Justina> cout "==", fmt(1234, HEX_U, FLAG_0X | FLAG_LEFT), "=="
==0X04D2  ==
Justina>
```

Print floating point numbers (integers are first converted). The print field width (10 characters) is extended if not wide enough to print all characters. 'FMT_NONE' resets all flags.

```
Justina> cout "==", fmt(12, 10, 3, EXP, FLAG_NONE), "=="
== 1.200e+01==
Justina> cout "==", fmt(12, 6, EXP), "=="
==1.200000e+01==
Justina> cout "==", fmt(12, 3, FIXED), "=="
==    12.000==
Justina> cout "==", fmt(12, 6, FIXED), "=="
== 12.000000==
Justina> cout "==", fmt(12.90, FIXED), "=="
== 12.900000==
Justina> cout "==", fmt(12.90, SHORT), "=="
==      12.9==
Justina>
```

Output a string (the notation is ignored if provided).

```
Justina>
Justina> cout "==", fmt("abc", 10, 4), "=="
==       abc==
Justina> cout "==", fmt("abcdef"), "=="
==      abcd==
Justina> cout "==", fmt("abcdef", CHARS, FLAG_LEFT), "=="
==abcd      ==
Justina> cout "==", fmt("abcdefABCDEF", 4, 6), "=="
==abcdef==
Justina> cout "==", fmt("abc"), "=="
==abc ==
Justina>
```

The following functions are useful to help formatting output.

tab ([n])	1. As argument of a print command: starting at the current column position, the *tab()* function inserts enough spaces to move to the next tab stop. If 'n' is specified, moves to the n-th next tab stop instead (starting from the current position). **<u>Do not</u>** include the *tab()* function in an expression. 2. Outside a print command, the *tab*() function returns the set tab size (optionally multiplied by 'n', if entered as argument).
col (n)	1. As argument of a print command: starting at the current column position, the *col()* function inserts enough spaces to move the print position to column 'n'. No spaces will be inserted if 'n' is less than or equal to the current print position. **<u>Do not</u>** include the *col()* function in an expression. 2. Outside a print command, *col*(n) simply returns 'n'.
pos ()	1. Within a print command: returns the column number where the print command started printing its first argument. The *pos()* function is useful when it is part of an expression. See the last example below. 2. Outside a print command: *pos*() returns the column number where the next print command for the stream last printed to, will start printing.

Change the tab size with this command:

tabSize n ;	Sets the distance between tab stops. Tab size is limited to a number between 2 and 30. Entering a number outside this range will set the tab size to one of these values without producing an error.

Notes

- The leftmost column is numbered '1'.
- Justina maintains current column positions separately for each individual external IO device and any open SD card file.
- The *tab()* and *col()* functions <u>work only if entered as direct arguments of print commands</u> - not if entered as part of an expression and not outside print commands.
- The pos() function is not affected by printing to variables.

Examples

Use of *tab()* function

```
Justina> cout "one", tab(), "two"; cout tab(), "three"
one     two     three
Justina>
```

The three words are printed at each tab stop.

Use of *fmt()* function together with *col()* function

```
Justina> coutLine fmt(12, 8, 2, FIXED), col(12), fmt(65.43)
   12.00      65.43
Justina>
```

The second value ("+++") starts printing at column 10, with the same formatting as the first number.

A more complicated example

In this example, we'll use the *fmt()* and *pos()* functions to obtain overlapping print fields.

Locate file 'overlap.jus' in folder '*libraries\Justina_interpreter\extras\Justina_language_examples*' (residing in your Arduino sketchbook location), and load the Justina program this file contains, using the procedure that was explained in chapter 2: *Getting started*.

We will not study this program here (programming will be discussed in chapter 13: *Programming*), but it may be interesting to have a look at the output.

The program contains two small functions (see next page); they both produce the same result. The purpose is to print 10 lines with each time two numbers of variable length, after each number a separator, and to fill up the remaining columns until column 15 with '+' characters.

Run the first function: type `ovlap1();` (+ Enter).

Then, run the second function: type `ovlap2();` (+ Enter).

Result:

```
Justina> ovlap1()
2.72---118 ++++
7.39---69 +++++
20.09---41 ++++
54.60---24 ++++
148.41---14 +++
403.43---8 ++++
1096.63---4 +++
2980.96---2 +++
8103.08---1 +++
22026.46---1 ++
                                                 0
Justina> ovlap2()
2.72---118 ++++
7.39---69 +++++
20.09---41 ++++
54.60---24 ++++
148.41---14 +++
403.43---8 ++++
1096.63---4 +++
2980.96---2 +++
8103.08---1 +++
22026.46---1 ++
                                                 0
Justina>
```

Both functions contain 2 print commands (**cout** and **coutLine**) each printing a part of each line.

Function ovlap1*()*: the **cout** command prints the two variable length numbers. The **coutLine** command then uses function *pos()* to determine how many '+' characters need to be printed.

```
function overlap1();
    var i=0, count=0, atColumn=0;

    for i = 1, 12;
        cout exp(i), "---", cInt(1.7 ** (10-i)), " ";
        coutLine repeatChar( "+", max(1, 16 - pos()));
    end;
end;
```

Extract of Justina program 'overlap.jus', edited in Notepad++ with the Justina language extension installed.

Function ovlap2*()*: the **cout** command only prints the first variable length number on each line. The **coutLine** command prints the second number, so, the length of the second number printed, which is stored in variable 'count' needs to be added to *pos()* to determine how many '+' characters need to be printed.

```
function overlap2();
    var i=0, count=0;

    for i = 1, 12;
        cout exp(i), "---";
        coutLine fmt(1.7 ** (10-i), 0, 1, "d", 0, count), " ",
            repeatChar( "+", max(1, 15 - (pos() + count)));
    end;
end;
```

Extract of Justina program 'overlap.jus', edited in Notepad++ with the Justina language extension installed.

Note: function '*repeatChar();*' (repeat character) is used in both examples to print the required number of '+' characters.

Also note that the '0' value displayed in the output is the Justina function result. Because no result was explicitly returned by the function (using the '**return** *expression'* command statement) zero was returned.

10.3 Reading from a stream

The functions below read one or multiple characters from a stream (external IO or SD card file if an SD card is available).

Most of the functions described in this chapter time out after a period that can be set by the user (see function *setTimeOut()*. During execution of these functions, system callbacks (if enabled – see *Appendix D: 'Running background tasks: system callbacks')* <u>continue</u> to be executed regularly (e.g., to maintain a TCP connection), so these functions can safely be used, for instance when reading a line of text from a remote TCP IP device.

Reading from the console

Note: any ***external*** IO device can be set as console stream (see further).

cin () *cin* ([terminator,] length)	Form 1: reads one character from the console. Returns the ASCII code of the character received as an integer. If no character is available, immediately exits, returning 0xFF (255). Form 2: reads characters from console until 'length' characters are read **or** terminator character (first character of a terminator string passed) is encountered **or** a timeout occurs. Returns a string with the characters read, or an empty string if nothing was read. The terminator character is <u>not</u> stored.
cinLine ()	Reads characters from console until the internal buffer is full **or** a newline character (0x0A) is encountered **or** a timeout occurs. Returns a string with the characters read or an empty string if nothing was read. The newline character, if encountered, is <u>added</u> to the string.

Example

In this example, *cin()* reads characters the user inputs and displays the corresponding ASCII codes. It does so until a 'q' is encountered.

We don't need to write a program to test this: create 2 variables, c and i (`var c, i;` + ENTER)

Then copy the line of code below, paste it into the command line and press ENTER.

```
i=1; while i; c= cin(); if (c<255); cout c, ", "; end; if (c == asc("q"));…
                                                            …i=0; end; end;
```

Now, type `abc` (+ ENTER)

Result:

```
Justina> i = 1; while i; c = cin(); if (c < 255); cout c, ", "; end; if (c == asc(
97, 98, 99, 13, 10,
```

The ASCII codes for characters a, b and c are printed (97, 98, 99) , followed the ASCII codes for the carriage return / line feed sequence (13, 10) as a result of pressing ENTER.

Press 'q' to quit and return to the Justina prompt.

Reading from any stream

These functions take one additional argument: a stream number. Constants IO1 to IO4 (or negative numbers -1 to -4) refer to an external IO device (Serial, TCP, LCD screen...), positive numbers to an open SD file.

Apart from the stream number (first argument) the syntax of these functions is identical to the syntax of the console read functions.

read (streamNumber) *read* (streamNumber, [terminator,] length)	Form 1: reads one character from the indicated stream. Returns the ASCII code of the character received as an integer. If no character is available, returns immediately, returning 0xFF (255). Form 2: reads characters from the indicated stream until 'length' characters are read **or** terminator character (first character of a terminator string passed) is encountered **or** a timeout occurs. Returns a string with the characters read, or an empty string if nothing was read. The terminator character is <u>not</u> returned.
readLine (streamNumber)	Reads characters from the indicated stream until the internal buffer is full OR a newline character (0x0A) is encountered **or** a timeout occurs. Returns a string with the characters read or an empty string if nothing was read. The newline character, if encountered, is <u>added</u> to the string.

Reading argument lists from a stream or variable

These functions read and parse a *<u>comma separated</u>* list of values (numbers and strings) from a stream or a string variable into a series of variables.

This offers a convenient way to safely read back and parse comma separated lists, created earlier with commands **coutList**, **printList** and **vprintList**, especially when working with SD files (although this works for any stream).

cinList (variable1 [, variable2, ...])	Reads a string from the console and parses the contents of that string into a list of variables. Reading stops when a newline character (ox0A) is encountered **or** a timeout occurs. Returns the number of variables that successfully received a value.
readList (streamNumber, variable1 [, variable2, ...])	Reads a string from the indicated stream and parses the contents of that string into a list of variables. Reading stops when a newline character (ox0A) is encountered **or** a timeout occurs. Returns the number of variables that successfully received a value.
vreadList (listVariable, variable1 [, variable2, ...])	Parses a string stored in 'listVariable' into a list of variables. Returns the number of variables that successfully received a value.

Notes:

- Receiving scalar variables will always store parsed values with the correct type (integer, float, string). Array variables have a fixed type and an execution error will occur if the value cannot be converted to the type of the array. Exception: floats will be converted to integers if required and vice versa.
- If the variable list does not contain enough receiving variables to store all values read, the rest of the values will be discarded. If the list contains more variables than required, then the extra variables will be left unchanged.

Example

Create scalar variables s, a, b, c and d first. Then, execute these statements:

```
s = "123, 456, \"abc\", 789";
vreadList (s, a = 0, b = 0, c = 0, d = 0);
a;
vprintList s="hello", a, b, c, d;
s;
```

Result:

```
Justina> s = "123, 456, \"abc\", 789"
                              123, 456, "abc", 789
Justina> vreadList(s, a = 0, b = 0, c = 0, d = 0)
                                                 4
Justina> a
                                               123
Justina> vprintList s = "hello", a, b, c, d
Justina> s
                              123, 456, "abc", 789
Justina>
```

This back-and-forth mechanism is safe for strings too, **even if strings contain backslash or double quote characters**.

Look for a character sequence ('target string') within a stream

These functions read characters from a stream (external IO or SD card file if an SD card is available) until a specific character sequence is found.

find (streamNumber, target string)	Reads characters from the indicated stream until the target string is found **or** a timeout occurs. The characters read are **not** returned: the function returns 1 if the target string was found, otherwise the function returns zero.
findUntil (streamNumber, target string, terminator string))	Reads characters from the indicated stream until the target string is found ***or*** terminator string is encountered ***or*** a timeout occurs. The characters read are **not** returned: the function returns 1 if the target string was found, otherwise the function returns zero.

Example

Execute this statement from the command line.

```
while !available(CONSOLE); end; find (CONSOLE, "abc"); coutLine  cinLine();
```

The command is echoed next to the Justina prompt, but a new prompt is not printed. This is because of the ' ! *available*(CONSOLE) ' expression: it checks whether characters sent from the console are waiting to be read, and as long as there aren't, it keeps waiting (returning FALSE, changed to TRUE by the negation operator).
Finally, the 'coutLine *cinLine()*' function at the end will capture any remaining characters and print them.

Now, you have the time to enter whatever text, press ENTER and check out the result.

Enter this: `I like Justina very much` and press ENTER

Result:

```
Justina> while !available(CONSOLE); end; find(CONSOLE, "abc"); coutLine cinLine()
                                                 0
Justina>
```

The function result is 0, meaning the string did not contain the target string "abc". Note that this function times out when the target string ("abc") is not found in the input: the Justina prompt will only appear after a short delay (that can be set).

Now, do this exercise again, but with a different text:

```
while !available(CONSOLE); end; find (CONSOLE, "abc"); coutLine cinLine();
Here you'll find the abc of Justina
```

Result:

```
Justina> while !available(CONSOLE); end; find(CONSOLE, "abc"); coutLine cinLine()
 of Justina
                                                 1
Justina>
```

The remaining text is printed and the function returns 1, indicating the target string was found.

10.4 Other stream functions and commands

The functions in the next table are mainly Justina wrappers to make the corresponding Arduino functions available to Justina. Console is the default device (in case no device number is specified).

peek ([streamNumber])	reads one character from a stream but does not advance to the next character. Function returns 0xFF (255) if no characters are waiting to be read
available (streamNumber)	Returns the number of characters waiting to be read.
flush (streamNumber)	Waits until all characters in output buffer have been sent. For instance, when writing to an SD card file, wait until all characters have been physically saved on the SD card (see next chapter).
setTimeout (streamNumber, timeout)	Sets the maximum time to wait for incoming characters, in milliseconds. When Justina starts, the default is set to 200 milliseconds. **Note that all functions reading data from external IO streams are impacted by this setting**, with only two exceptions: *cin*() and *read*(stream), called without arguments.
getTimeOut (streamNumber)	Returns the timeout set for a stream, in milliseconds.
availableForWrite (streamNumber)	Returns the number of characters that can be written to the output buffer for a stream without introducing delays.
getWriteError (streamNumber)	Returns the last error generated by a stream write operation.
clearWriteError (streamNumber)	Clears any write error.

Commands to change streams designated as console and debug out streams

The following commands let you set the console or the 'debug out' stream to another I/O device or('debug out' only) open file number.

☞ The console refers to the input/output device sending user input (user commands, ...) to Justina and / or receiving standard Justina output (calculation results, error messages...). Example: the Arduino IDE serial monitor.

setConsole streamNumber ;	Changes the console to another external IO device. use: command line only.
setConsoleIn streamNumber ;	Changes the console to another external device for input but keeps the currently assigned device for console output. use: command line only.
setConsoleOut streamNumber ;	Changes the console to another external device for output but keeps the currently assigned device for console input. use: command line only.
setDebugOut streamNumber ;	Changes the debug output stream to any external IO device <u>or open file number</u> (see next chapter). use: command line only.

⚠ Avoid changing the console to a stream which is currently unavailable (for instance a TCP terminal that is currently offline). Appendix D: '*Running background tasks: system callbacks*' discusses a method to recover from such a situation.

Print a list all variables

Sometimes it's handy to get an overview of all created variables with their type and current value:

listVars [streamNumber] ;	Print a list of all user and global program variables to the indicated device number or open file number. The 'streamNumber' argument can refer to any available external IO device **and** to any open file. If no device number is specified, prints to console.

The variables (and constants) are listed in the order created but in two groups: user variables first. In each group constants are printed on top.

Information printed includes variable name, type, constant or variable ad value.

User variables have an extra column 'U' (used): an 'x' is printed if a user variable is referenced by the currently loaded program (because the program didn't define a variable with this name). This is quite useful to retain specific data after the program has been cleared (or replaced by another program).

Notes

☞ A user variable in use by a program cannot be deleted as long as the program is loaded.
☞ a program cannot be loaded if variables it references are not defined (not as program variable and not as user variable).

Example

```
Justina> listVars

user variable                 U type    qual   value
-------------                 - ----    ----   -----
j                               float   const  2.72
t12345                        x float          0.00
abcde                           string         "hello"
p12                             float          (array 10 elem)
p13                             float          (array 20 elem)

global prog variable            type    qual   value
--------------------            ----    ----   -----
factors                         string         (array 3x5 = 15 elem)
abcde                           integer        123

Justina>
```

In this example, the program currently loaded has a variable named 'abcde'. A user variable with the same name exists as well. They live together peacefully; however, the Justina program can only access its own (integer) variable, which makes the user variable inaccessible. Vice versa, the user can only access the (string) user variable.

But, except for these two variables named 'abcde', the program can access all user variables and a user can access all global program variables.

The 'x' in the 'U' (Used by program) column tells us that the user variable 't12345' is being referenced in the current program (the program uses this variable because it hasn't defined a program variable with that name).

11 Working with SD cards

Connecting a micro-SD card reader to the Arduino opens up a whole new world: you can

- create files on your SD card, write data to / read data from files
- receive and send files from / to your computer or any other device (possibly another Arduino)
- load programs from an SD card instead of loading them from your computer via USB
- enable an AutoStart function: automatically load and execute a startup file as soon as Justina is launched (for instance to select specific display settings, angle mode etc.)
- ...

Justina works internally with the **Arduino SD card library** but this is completely transparent to the user.

This library uses the older 8.3 file format (max. 8 characters for the file name, 3 characters for the extension, filenames are not case-sensitive) which is more than sufficient for our needs.

Only SD cards with a **maximum size of 32 Giga Byte** are supported. The Arduino SD card library only supports SD cards formatted for FAT16 and FAT32 file systems (or with a partition formatted as FAT16 or FAT32).

Note for users working with the Arduino nano ESP32 board

The nano ESP32 does not use the standard Arduino SD library, but an SD library specific for this ESP32 board.

Although the command set and functions are more or less identical, the nano ESP32 library has a few restrictions as compared to the standard Arduino SD library. In a nutshell:

- you cannot open a file in a combined read/write (or read/append) mode. After writing (or appending) to a file, you need to close it and reopen it for reading
- when opening a file for writing (not for appending), the file is automatically truncated before you start writing to it.
- in write mode, the *size()* function does not return the current size; as soon as you start writing it will return 0.

Connecting an SD card breakout board to your Arduino

Micro SD card readers use a standardized interface, requiring only 5 connections between Arduino and card reader: GND (ground), Vcc, Clock (CLK), Data In (DI), Data Out (DO) and Chip Select (CS).

Within Arduino, communication is handled by means of the SPI library. This requires the use of specific Arduino pins to connect to the SD card breakout box pins, with one exception: **by default**, when the Justina object is created, the **Chip Select pin is set to Arduino pin 10**. To select another pin as CS pin: please refer to Appendix A: *Creating a Justina object and choosing* startup options.

Detailed instructions on how to connect and test an SD card are outside the scope of this manual, but if not yet familiar with the process of hooking up an SD card to your Arduino, this might be a good time to familiarize yourself with it. You'll find a good introduction in this article: https://learn.adafruit.com/adafruit-micro-sd-breakout-board-card-tutorial . You'll also find there how to test whether your card is working.

An example of a Micro SD card breakout board: https://www.adafruit.com/product/254 .

11.1 Starting Justina with an SD card mounted in its SD card slot.

By default, when the Justina object is created, SD card functionality is enabled but the SD card (if present) is not yet started, or 'mounted' (to mount the SD card automatically, and even run a Justina autostart program file if desired, or to disable starting the SD card all together, please refer to Appendix A: *Creating a Justina object and choosing* startup options).

Example

- If required: on your computer, if needed, format an SD card (maximum size **32 GB, FAT16 or FAT32 format**).
- insert the SD card in the SD card slot (preferably when the power is off) and start Justina.

Now, type in these statements (each time pressing ENTER)

`startSD ;`	(this command does nothing if the SD card was started already)
`listFiles ;`	

Result (in this example, the SD card contains data):

```
Justina> startSD
Justina> listFiles

SD card: files (name, size in bytes):
System Volume Information/
  WPSettings.dat              12
  IndexerVolumeGuid           76
factorial.jus                 1714
input.jus                     2488
SD_parse.jus                  5141
SD_test.jus                   4470
```

11.2 SD card functions and commands

This chapter describes all functions and commands dedicated to working with files and directories.

To read or write a file, you must first open it. Open files (and directories) are referred to by a file number assigned to the file when opening the file.
File and directory names always include the full file path. Use '/' (slash) characters as separators in file paths. The leading "/" is optional.

Open files represent streams; file numbers are associated with open files. The maximum number of open files is 5; the file numbers are always in the range 1 to 5 (predefined constants FILE1 to FILE5 – see previous chapter: *Input and output*).

Most commands and functions with a device number as one of the arguments (e.g., **'printLine'**) accept file numbers as well as device numbers, as indicated in the documentation for these respective commands and functions (see previous chapter: *Input and output*).

Referring to a file number without an associated open file will always produce an error.

1. Open files are referred to by file number, closed files are referred to by file name.

Functions for working with SD cards and files:

<table>
<tr>
<td>open (filename, mode)</td>
<td>Opens the file with the specified filename and returns a file number assigned to the open file.

filename: name in 8.3 file format: 8 characters maximum and a 3-character extension, preceded by the full path.

mode: specifies the access mode by using the predefined constants listed below. Constants may be combined using the bitwise ‘or’ operator (|).

READ 1 open file for reading (this is the default)
WRITE 2 open file for writing
APPEND 6 open file for writing; data will be appended to the end of the file

<u>Note</u>: combining WRITE and APPEND constants will set APPEND mode

NEW_OK 16 if file doesn't exist, create and open a new file
NEW_ONLY 48 only create and open new files, do not allow opening existing files

TRUNC (**) 64 on opening, delete all existing file content (empty the file)

SYNC (**) 8 synchronous writes: send data physically to the card after each write (minimize data loss after a crash)

** Notes for users of the nano ESP32 board: files will always be truncated when opening a file for WRITE. Because of restrictions of the underlying SD library for ESP32, constants TRUNC and SYNC have no effect when working with the nano ESP32 board.</td>
</tr>
</table>

close (fileNumber)	Close the file associated with the file number. If the file is not open, an error is produced.
position (fileNumber)	Returns the current position in the open file: the location in the file where the next byte will be read or written. The first byte in the file has position 0.
size (fileNumber)	Returns the file size (in bytes) of an open file **Note for users of the nano ESP32 board**: when a file is opened in WRITE or APPEND mode, *size()* does not return the actual size of the file.
seek (fileNumber, position)	move to a specific position in the file. Position argument: zero will move to beginning of file; constant EOF will move to end of file (after the last byte in the file). **Note for users of the nano ESP32 board**: use *seek()* in READ mode. In WRITE or APPEND mode, this function does not allow you to freely move the insertion point (only sequential writing).
name (fileNumber)	returns the filename of the file, without path
fullName (file number)	returns the name of the file, including the full path
isDirectory (fileNumber)	returns 1 if the open file is a directory, otherwise returns zero
rewindDirectory (dirFileNumber)	Used together with function openNextFile*()*, to go back to the first file in a directory. returns 0.
openNext (dirFileNumber)	opens the next file in the open directory associated with dirFileNumber and returns the file number of the open file. When all files in the directory have been opened, the next call to function *openNext()* will return zero (no error will be produced). Note that the previously opened file won't be closed automatically when you open the next one: it must be closed explicitly (either before or after opening the next file)
isInUse (fileNumber)	Returns 1 if a file associated with this file number is open, otherwise returns zero.
closeAll ()	Close all open files. Always returns zero.
exists (fileName)	The function returns 1 if a file (or directory) with the specified name (including full path) exists and returns zero otherwise. If the name is not a valid name, an error will be produced.
createDirectory (dirName)	Create a directory with the specified name (including full path). If the parent directory is not the root ("/") directory, then any missing subdirectories along the path will be created as well. The function will return 1 if success and will return zero if the directory cannot be created (e.g., because the directory already exists). If the name is not a valid name, an error will be produced.
removeDirectory (dirName)	Remove the directory with the specified name (including full path). The parent directory will <u>not</u> be removed, even if becoming empty. The function will return 1 if success and will return zero if the directory cannot be removed (e.g., because the directory does not exist). If the name is not a valid name, an error will be produced.

remove (filename)	Remove the file with the specified filename (including the full path). The function will return 1 if success and will return zero if the file cannot be removed (e.g., because it's currently open).
fileNum (filename)	If the file with the specified filename is open, return its file number, otherwise return zero. If the name is not a valid name, an error will be produced.

Commands available for working with SD cards and files:

startSD ;	Initialize the SD card. When creating the Justina object in your Arduino sketch, you can select this to happen automatically when Justina is started, or not.
stopSD ;	Send all data not yet physically sent to the SD card (including updating file structure etc.), close all open files and 'stop' the SD card. Note that this action is always performed automatically when quitting Justina. **Before removing or inserting a card while Justina is running, always execute command 'stopSD'. If not, you do not only risk losing data but your Arduino may hang.**
receiveFile streamNumber, filename [, verbose] ; - or - **receiveFile** fileName ;	Receive a file from an IO device and save it on the SD card, with the specified filename. The default device number is CONSOLE. If verbose is FALSE (‘off’), overwrite existing files without warning and do not print any other messages (e.g., when using this command from within a program). The default is TRUE (verbose ‘on’). - or - Receive file from the console and save it on the SD card with the specified filename. Verbose is ‘on’. Upon execution of this command, Justina will wait 10 seconds for the first character to arrive, giving you time to start transmission at the other end (e.g., your computer). It will stop receiving characters after a set timeout is reached (see *setTimeout*() function. Note: Justina can not only receive text files (e.g., Justina program files, data files organized as records with text fields, ...) but binary files as well (e.g., an image file).
sendFile filename [, streamNumber [, verbose]] ;	Sends the file with the specified filename to an IO device. The default device number is CONSOLE. Verbose: if FALSE, output no messages (e.g., when using this command from within a program. The default is TRUE.
copyFile sourceName, destName, [, verbose] ;	Copy a source file to a destination file. Verbose: if FALSE, overwrite existing files without warning and do not print any other messages (e.g., when using this command from within a program). The default is TRUE.
listFiles [streamNumber] ;	Print a list of all files and directories on the SD card and send the output to the indicated device number. For each file, the filename and the size are printed. The device number can refer to any available external IO device **and** to any open file. If the device number is not specified, the list is printed to the console.

listFilesToSerial ;	This is simply a 'wrapper' around the SD library method to print a list of files. This always prints to Serial (even if Serial is not the console) but it includes a 'date and time' field (which is only helpful if a real time clock is present to supply the correct time - which is not implemented in the current version). **Note for users of the nano ESP32 board**: this command is not available.

Example

In this example we'll open a file with filename 'myFile' for writing data to it. If the file doesn't exist yet, it must be created, but if the file does exist, its current contents must be deleted upon opening.

Create a variable myFileNum and then type in this statement (+ ENTER)

```
myFileNum = open("myFile", WRITE + NEW_OK + TRUNC);
```

This will open file "myFile" for WRITE, truncating its contents. If the file does not exist, it will be created.

The file number assigned to the now open file is stored in variable 'myFileNum'

Now let’s add 3 text lines to the file, with the same **'printLine'** statements we used earlier to send characters to the console (or any other external IO device) and close the file:

Type in these statements (each time pressing ENTER)

```
printLine myFileNum, "this is line one";
printLine myFileNum, "this is line two";
printLine myFileNum, "this is line three";
close(myFileNum)
```

Result

```
Justina> myFileNum = open("myFile", WRITE + NEW_OK + TRUNC)
                                                  1
Justina> printLine myFileNum, "this is line one"
Justina> printLine myFileNum, "this is line two"
Justina> printLine myFileNum, "this is line three"
Justina> close(myFileNum)
                                                  0
```

The file number assigned to the file is 1.

Example

We'll reopen the file we just created for reading, read some of its contents, use the *position()* and *seek()* functions and finally close the file again.

Type in these statements (each time pressing ENTER)

`myFileNum = open("myFile", READ);`	
`readLine(myFileNum);`	
`position(myFileNum);`	
`read(myFileNum, 13);`	read 13 characters
`position(myFileNum);`	
`read(myFileNum, "l", 20);`	read until an 'l' is found in next 20 characters. (The 'l' itself is not returned.)
`readLine(myFileNum);`	
`seek(myFileNum, 18);`	
`readLine(myFileNum);`	
`close(myFileNum);`	

Result:

```
Justina> myFileNum = open("myFile", READ)
                                                    1
Justina> readLine(myFileNum)
                               this is line one

Justina> position(myFileNum)
                                                   18
Justina> read(myFileNum, 13)
                                    this is line
Justina> position(myFileNum)
                                                   31
Justina> read(myFileNum, "l", 20)
                                    two
this is
Justina> readLine(myFileNum)
                                      ine three

Justina> seek(myFileNum, 18)
                                                    0
Justina> readLine(myFileNum)
                               this is line two

Justina> close(myFileNum)
                                                    0
Justina>
```

Example

Let's now send the contents of this SD card file to the console. Type `sendFile "myFile", CONSOLE;` (+ ENTER)

```
Justina> sendFile "myFile", CONSOLE

Sending file... please wait
this is line one
this is line two
this is line three

+++ File sent, 56 bytes +++

Justina>
```

Example

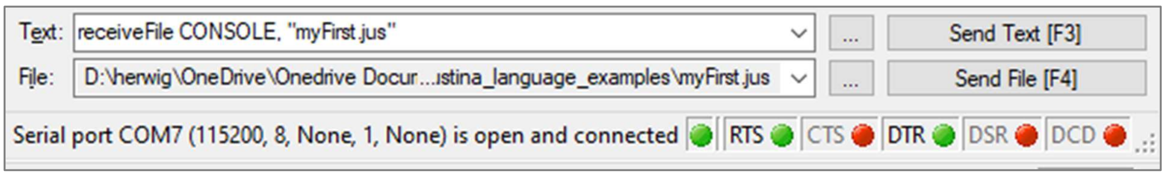
In chapter 2: *Getting started* we created a small Justina program and saved it on the computer. Then, using YAT as Terminal Application, we sent this file to the Arduino, where it was parsed immediately (**'loadProg'** command), ready to run. But now, we have an SD card. So, why not directly load a program straight from the SD card ?

Of course, on the computer, we could copy the file to the SD card and then place the card in the Arduino SD card slot. But it's much easier to send the file straight from the computer to the SD card.

We will again use YAT Terminal:

- Using the button with 3 dots to the left of YAT key 'send file', select file 'myFirst.jus' (but don't send it yet).
- Type and execute statement '`receiveFile CONSOLE, "myFirst.jus";`' in the command line (as CONSOLE is the default, this argument is optional). This instructs Justina to start waiting for a file, listening to the 'console input' stream (make sure to choose a file name complying with the 8.3 file format, if not, you'll get an error).

- Send the file you just selected to Arduino (YAT button 'Send File'). It will be saved on the SD card.



Now, load the program straight from the SD card: type and execute `loadProg "myFirst.jus";`

(The complete syntax of **loadProg** is discussed in next chapter: *Other functions and commands*).

Finally, execute the only function contained in the program: `print5lines();`

The console output now looks like this:

```
Justina> receiveFile CONSOLE, "myFirst.jus"

Waiting for file...
Receiving file... please wait

+++ File received, 1402 bytes +++

Justina> loadProg "myFirst.jus"

Loading program...

Program parsed without errors. 0 % of program memor

Justina> print5lines()
line = 1
line = 2
line = 3
line = 4
line = 5
                                        36.00

Justina>
```

Notes:

- the filename on the SD card is unrelated to the filename on the computer.
- if a file with that filename exists already on the SD card, Justina will ask your permission to overwrite it.

12 Other functions and commands

Loading and clearing a program, clearing all of memory

Before you can execute a program, you must load it into memory.

Loading a program is the process of reading a source file, parsing it into a sequence of tokens, storing tokens in program memory and creating and initializing program variables. When a program is launched, tokens will then be read by Justina and executed.

When an execution error occurs and for debugging purposes, statements can be 'unparsed' with the help of extra information stored separately in order to give meaningful messages to the user.

When a program is loaded, any previous program, with associated program variables, is first removed. This does not affect user variables, which remain available and keep their values.

loadProg [streamNumber] ; - or - **loadProg** fileName ;	Receive a Justina source file from an external IO device (if no argument is provided, receive from console) - or - read a Justina source file with the specified filename from the SD card. Parse the source file into tokens. If receiving the Justina source file from an external IO device (e.g., the console), Justina will wait 10 seconds for the first character to arrive, giving you time to start transmission at the other end (e.g., your computer). It will stop receiving characters after a set timeout is reached (see *setTimeout*() function). Use: command line only.

Commands for clearing (part of) memory:

clearProg ;	Remove a parsed program from program memory (with associated program variables). This command is non-executable (it is parsed but is skipped during execution), however it will take effect only after execution has ended. Use: command line only.
clearMem ;	Clear all: same as **clearProg** but remove all user variables as well. This command is non-executable (it is parsed but is skipped during execution), however it will take effect only after execution has ended. Use: command line only.

Quitting Justina

quit ;	Quit Justina and return to the calling Arduino program (right after the begin*()* method). On quitting, Justina is kept in memory (including the currently parsed program, program and user variables, settings etc.). If the Justina begin*()* method is called again at a later moment, you can *continue* your work, right where you left off, without any loss of data. To completely remove Justina from memory, delete the Justina object (in your Arduino program).

13 Programming

To write and execute Justina **programs**, you'll need a text editor on your computer. It's highly recommended to use **notepad++** (free): it displays line numbers (important once you start debugging a Justina program) and it has Justina syntax highlighting, which is invaluable when editing larger Justina programs. A Justina 'User Defined Language file' is available in the Justina library for that purpose.

Second, you'll need a terminal program to send your program to your Arduino. Unfortunately, the Serial Monitor of the standard Arduino IDE is not capable to send files to the Arduino. As already mentioned, a good choice is **YAT** (free).

As you may have noticed in previous chapters, notepad++ and YAT are used throughout this manual in examples.

So, if not already done so, you might want to install these two applications right now. For installation instructions, please refer to Appendix F: *Installing Notepad++ and the Justina language extension,* and Appendix G: *Installing YAT terminal*.

Within a program, all statements must be separated by a semicolon (;). A source line can contain multiple statements and statements can span multiple lines.

13.1 Program and program functions

Preliminary note: all identifier names (program name, function names, variable names) must follow the same naming convention: names must start with a letter from a to z (or A to Z), and may be followed by a sequence of letters, digits and underscore characters. The maximum name length is 20 characters. Names are **case sensitive**.

Every program must start with a **program** statement, giving the program a name. This should be the first statement in your program file (excluding comments).

Program programName ;	Marks the start of the program and gives the program a name. Must be the first statement in the program file (only to be preceded by comment lines). Please note that the name is <u>not</u> used to start a program, only to label it.

<u>Functions</u>

To be meaningful, a program must contain at least one function. A function starts with a **function** statement, and it ends with an **end** statement. These two statements mark the 'physical' start and end of the function.

The function statement specifies the function parameters (values that can be passed to the function or returned to the calling function) and attributes a function name to the function.

function functionName ([param name [()] , param name [()] , ...] , [param name = literal , ...]) ; [statement; statement; ...] **end ;**

A function may receive scalar values as arguments as well as complete arrays.

- A parameter name followed by empty parentheses indicates that an array is expected as argument. Without the parentheses a scalar is expected.

Function parameters can be either mandatory or optional.

- Optional parameters are followed by an equal sign and a literal, forming an initializer which serves as default value for the function parameter in case the calling function does not supply an argument. Optional parameters always expect scalars as arguments. All mandatory parameters must precede the optional parameters.

Example

```
Function volumes ( length, manyWidths(), height, id="zzz", unit=2 );
... (function body)
end;
```

Function 'volumes' has three mandatory parameters (the second one being an array), followed by two optional parameters. If optional arguments are not provided, the function parameters will receive the initial values "zzz" and 2, respectively.

Calling a function

To call a Justina user function, use the syntax used for calling an internal Justina function, like '*sin()*' etc.

The function name to call is followed by a list of arguments corresponding to the list of function parameters in the definition. Where an array is expected, enter the array name (without parentheses), where a scalar is expected, enter an expression, variable or constant.

function name ([expression or array name, expression or array name, ...])

Supply all mandatory arguments. Optional arguments can be left out, as desired. If an optional argument is left out, next optional arguments must be left out as well.

Variables (scalars and arrays) are always passed by reference. That means that the called function will not make local copies of passed variable values but will store a reference to the variables instead (scalar, array element or array). Any changes made within the called function will be reflected in the original variables.

Constants and the results of expressions are passed by value.

If the constant is a string constant, the 'value' passed will be a reference to the character array where the string is stored - individual characters are never passed (for a string variable, a reference to this 'value' is passed).

Example

`var length=5, widths(10)=2, height=10;`	
`widths(8)=6;`	
`volumes ( length, widths, (height), "abc");`	call function *volumes()*

Scalar variable 'length' and array 'widths' will be passed by reference, expressions '`(height)`' and `"abc"` will be passed by value.

- if you don't want a called function to alter a variable, put the variable between parentheses (creating an expression)
- an array is always passed by reference

Functions may call other functions and they may even call themselves (this is called recursion; an example program is included in the library and we'll discuss it in a moment).

The function called from the command line (by the user) is the function where program execution starts, or main function (the program name itself is not used to start a program).

Returning control to the caller

return [expression] ;	Exit the current function and return control to the caller (the function that was calling the current function, or the command line if it was directly called from there). If 'expression' is present, it is evaluated and the result (integer, float, string) is returned to the caller as function result. If not present, zero is returned (integer value).
end ;	Exit the current function and return control to the caller (the function that was calling the current function, or the command line if it was directly called from there). Return integer value zero to the caller.

13.2 Variable declarations in a program

We already encountered the **var** and **const** commands when we discussed user variables and user constants. Within a program, they serve the same purpose, which is to create program variables and constants.

But the syntax is still the same:

var name1 [(dim1 [, dim2 [, dim3]]])] = literal1 [, name2 ...] ;

const name 1= literal1 [,name 2= literal2,...] ;

In a program, outside a function, ***var*** *and* ***const*** *declare global program variables and constants*

Global program variables can be referenced (used in equations, as arguments of a function, ...) anywhere in a program, with one restriction: a variable can only be referenced (e.g. used in an expression) within a program once it has been declared (after the declaration, further down the program), because the parser makes only one pass (it reads the source program file only once, from top to bottom) and it needs to know where memory has been allocated for a variable when it encounters a reference to that variable.

A global program variable / constant can be used in immediate mode as well (from the command line) unless it's 'shadowed' by a user variable / constant having the same name (scope).

Memory is allocated to global program variables (and constants) during parsing. It remains allocated until the program is deleted or overwritten by another program (lifetime).

Commands **var** and **const** are the only statements that may appear outside a function's body.

Within a function, ***var*** *and* ***const*** *declare local function variables*

Local function variables, as their name implies, are only 'known' inside the function where they have been defined. Also here, the rule applies that they can only be referenced (in the function) once they have been declared (further down the program file).

Memory for local variables (or local constant variables) is allocated - and variables receive their initial values - when a function is called, and before the function starts executing. Memory is deallocated when a function ends (control returns to the caller).

Within a function, ***static*** *declares static function variables*

Just like local variables, static variables are only accessible within the function where they are defined. And also here, the rule applies that they can only be referenced once they have been defined (further down the program file).

However, memory for these variables is not allocated when a function is called, but during program parsing (and that's also when these variables receive their initial values). Static variables are destroyed when a program is deleted or a new program is loaded (same lifetime as global program variables).

The syntax is identical to the **var** command syntax:

static name1 [(dim1 [, dim2 [, dim3]]])] = literal1 [, name2 ...] ;

Static function variables retain their values between successive calls of the function.

Notes

- All local and static variables without initializer are defined as float and initialized to zero during parsing. Same applies to all array elements of arrays without initializer.
- Variable declaration commands are non-executable commands, as their only purpose is to inform Justina of the existence of these variables / constants in order to reserve memory for them (during parsing or, for local variables, before a function is called). They are never executed. That means you can put them inside a loop for example (which is not necessarily good programming practice), they will then be available from that point onward until the end of the function.
- During the lifetime of global program variables (and constants), they are accessible from the command line by the user (except when 'shadowed' by user variables / constants with the same name).
- Vice versa, a program has access to user variables (unless shadowed by program variables).
- Within one function, a variable name can only reference one variable, be it a user or global program variable, a local function variable or a static function variable. Local and static function variables 'shadow' global variables with the same name.

Comments

A comment is any text that you add to your source file for documentation purposes and that should be ignored by the Justina interpreter. Two forms exist:

- Single line comment: anything between '//' (two slash characters in a row) and the end of a source line.
- Multiline comments anything between '/*' (slash and asterisk) and '*/" character sequences.

Comments do not need to start at the beginning of a line. Note that multiline comment blocks cannot be nested.

13.3 Control structures

Control structures are defined by specific statements controlling how execution should proceed. They decide how the flow of executed statements should be altered at certain moments.

In Justina, control structures always start with a specific control statement and they always end with a control statement (in Justina, that's always an **end** command). The statements in between form a statement block.

Control structures may contain other (nested) control structures. This is what makes a program structured, making a program much easier to develop and maintain with less possibilities to introduce program errors.

if...end structure

The 'if' control structure starts with the **if** command and ends with the **end** command. Optionally **elseif** and **else** commands can occur in between, creating multiple statement blocks.

if test expression ;	[statement; statement; ...]
[elseif test expression ;	[statement; statement; ...] **]**
...	
[elseif test expression ;	[statement; statement; ...] **]**
[else ;	[statement; statement; ...] **]**
	[statement; statement; ...] **]**
end ;	

Test expressions are evaluated one by one until a test expression returns a non-zero result (interpreted as TRUE). If that happens, the corresponding statement block is executed after which execution continues after the **end** statement. If all test expressions return zero (false), and an **else** clause is present, the statement block following the **else** clause is executed.

If a test expression returns a non-numeric result, an execution error is produced and execution stops.

for...end loop

Using the '**for**...**end**' control structure, the statement block in between can be executed multiple times: it defines a **for**...**end** loop. This is controlled by 'control variable' (a scalar or an array element).

```
for control variable [ =  start ] , end  [ , step ] ;
      [statement; statement; ... ] ;
end ;
```

'Start', 'end' and 'step' : numeric expressions yielding a numeric result (the default step = 1).

First, 'control variable' receives the value 'start' (if a start value is not specified, it maintains its current value).

Then the statement block is executed repeatedly. At the end of each iteration, the value of 'control variable' is incremented by 'step'. If this new value is still in the range between 'start' and 'end' values, a new iteration starts. Otherwise, the loop ends and execution continues after the **end** statement.

Notes

- to test the current value of 'control variable' after each iteration, 'end' and 'step' values will be converted to the type of 'control variable'
- if 'end' is higher than 'start' but 'step' is negative, then the loop will not be executed (and same if 'end' is less than 'start' but 'step' is positive)
- the value of 'control variable' should not be changed within the statement block
- two nested **for**...**end** blocks can not share the same control variable

while...end loop

Using the '**while**...**end'** control structure, the statement block in between can be executed multiple times: it defines a **while**...**end** loop.

```
while test expression ;
    [statement; statement; ... ] ;
end ;
```

First, 'test expression' is evaluated. If its result is not equal to zero (TRUE), the statement block is executed. At the end of each iteration, the test expression is evaluated again and if the result is still not equal to zero, the next iteration starts.

When the test expression result becomes zero (FALSE), the loop ends and execution continues after the **end** statement.

Other commands changing the program execution flow

break ;	This command ends execution of a loop. Execution continues after the loop **end** statement.
continue ;	The remainder of the currently executed loop is skipped, moving on immediately with the test at the end of the iteration. The test result determines whether a next iteration starts or the loop is ended.

Notes

- control structures can be used in immediate mode as well
- In Notepad++, using the Justina Language extension, the **function**...**end** structure, the **return** statement, the control structures and the **break** and **continue** commands are displayed in a bold and slightly darker color to distinguish them from other (non-control structure) commands

Example: Justina program ‘factorial’

Locate file ‘fact.jus’ in folder *‘libraries\Justina_interpreter\extras\Justina_language_examples\’, residing in your sketchbook location, and* open it in Notepad++.

The program calculates the factorial of a positive integer, using a recursive mechanism: the program repeatedly calls itself, until a final result is calculated.

```
28  program factorial; // this is a JUSTINA program
29
30  function fact(n);
31      var fact_n = 0;                           // local variable
32
33      n = cInt(n);                              // do nothing if int
34
35      if (n > 2);
36          fact_n =  n * fact(n-1);              // recursive call: c
37      else;
38          fact_n = n;
39      end;
40
41      return fact_n;                            // return n!
42  end;
```

We will execute this program when we discuss debugging, a little bit further down. But let’s try to find out how it works now.

This program has only one function, named *fact*. It has one parameter ‘n’, without initializer, which means that an argument has to be supplied when this function is called. It also has a local variable ‘fact_n’ which will store calculated factorials.

When function *fact()* is called from the command line with value ‘3’ as argument:

- local storage for variables ‘fact_n’ and ‘n’ (which receives value 3) is created and *fact()* starts executing
- to calculate 3! as 3 * 2! , 2! must be calculated first (**if** clause, line 22)
- to calculate 2!, *fact()* calls function *fact()* again, with ‘2’ as argument, creating a second independent instance of *fact()*
- local storage for variables ‘fact_n’ and ‘n’ (which receives value 2) is created and *fact()* starts executing
- *fact()* calculates 2! as 2 (**else** clause, line 24) and returns 2!
- instance 1 of *fact()* resumes and can now multiply 3 with 2! : it returns 3!

Note that this is not a very efficient way to calculate factorials. The higher the input value, the more instances of ‘*fact()*’ we need concurrently. At a certain moment RAM memory will be completely used (remember that Arduino is still a microprocessor, with a relatively small amount of memory and no means of effectively managing memory – notably the ‘heap’, where all local variables are stored) and the processor will simply hang before it could start releasing memory as function instances end.

The following lines of code do exactly the same thing, and we’re not even writing a program. You can do this from the command line:

```
var n=0, i=0, fact=0;                    // init as integer
n=1; fact=1; for i =2, n; fact=fact * i; end;     // 1!
n=4; fact=1; for i =2, n; fact=fact * i; end;     // 4!
n=6; fact=1; for i =2, n; fact=fact * i; end;     // 6!
```

13.4 Commands to interact with the user

A few commands allow the user to interact with a running program without stopping Justina background tasks (e.g. maintaining a TCP connection – see Appendix D: '*Running background tasks: system callbacks*').

input prompt, value, flag ;	Halts the program, displays a message on the console and waits for an answer from the user before proceeding.

prompt character string expression, which will be displayed as a message on the console.

value must be a **variable**. On entry, optionally contains a default answer (a string). On exit, will contain the answer entered by the user in response to the message. The answer is always stored a string. To cancel the input operation, type (or include) '\c' in the answer.
In order to select the default answer, type (or include) '\d' in the answer.

flag This mandatory argument must be a **variable**. Supplying a constant or an expression instead of a variable will lead to a runtime error. Possible values on entry:

NO_DEFAULT	0	selecting a provided default answer is not allowed.
ALLOW_DEFAULT	1	selecting a provided default answer is allowed.

on exit, argument 'flag' (a variable), will contain:

CANCELED	0	the user canceled the operation
OK	1	the user confirmed by pressing ENTER, or entered 'Y' as a valid

Note: all characters typed to form an answer are stored, including '\' characters (escape sequences are not processed). You don't need to type surrounding quote, because a string is what is expected.

info prompt [, flag] ;	Halts the program, displays a message on the console and waits for the user to choose between a few options before proceeding.

prompt character string expression, which will be displayed as a message on the console.

flag Optionally, supply a value for this argument. Note: if this argument is provided, it must be a **variable**. Supplying a constant or an expression instead of a variable will lead to a runtime error. Possible values on entry:

ENTER	0	confirmation required by pressing ENTER (other characters are ignored). This is the *default*.
ENTER_CANCEL	1	idem, but ' \c ' (cancel) is allowed as well in the input.
YES_NO	2	Only 'Y' or 'N' (yes or no) are accepted as answer.
YN_CANCEL	3	Only 'Y' or 'N' (yes or no) and ' \c ' (cancel) are accepted as input.

on exit, argument 'flag' (which must be a variable), if supplied, will contain the user answer:

CANCELED	0	the user entered 'cancel'
OK	1	the user confirmed by pressing ENTER, or entered 'Y' as a valid answer
NOK	-1	the user entered 'N' as a valid answer

Two other commands to interact with the user are useful as well.

pause seconds ;	Pause for a whole number of seconds. Pressing ENTER on the console keyboard will immediately resume execution. Minimum is 1 second. A floating-point argument will always be converted to integer first.
halt ;	Displays a message on the console ("Press ENTER to continue") and halts the program until ENTER is pressed on the console keyboard. Note: this command should not be confused with the '**stop**' command (see section 13.6: *Debugging*, below).

13.5 Error trapping

Normally, when an execution error occurs, Justina will display an error message and execution will end.

Example: if in a program, a statement `asin(-2);` is executed, execution will terminate and an error will be produced (-2 is not within the domain of the arc sine function):

```
 asin(-2)
 ^
 Exec error 3100 in user function fact, source line 11
Justina>
```

➢ Remark that the error message also indicates the Justina function and source line.

But there can be situations where, if an error occurs, we don't want the program to terminate. Instead, we want to test for errors and take appropriate action. Two commands and a function are provided for this.

trapErrors trap **;**	trap: numeric argument. If trap is not equal to zero, clear the last error and set error trapping on. If zero, set error trapping off (but do not clear the last error).
clearError ;	clear the last error
err([evalParseError])	If error trapping is enabled, *err*() returns the last execution error that occurred. If no error present, returns 0. Special case: if the execution error signals a runtime **parsing** error during execution of an *eval*() function or a list parsing function (*cinList*(), ...), the parsing error is returned to the 'evalParseError' argument (if provided), which must be a variable capable of storing an integer value.

The following command is not directly linked to error trapping, but we'll put it here:

raiseError number **;**	"produce" an error with the specified number. Justina will behave as if the error actually occurred.

Example: program ‘input’

In this example, we will use the input command together with the *eval()* function and error trapping.

Locate file ‘input.jus’ in folder *‘libraries\Justina_interpreter\extras\Justina_language_examples'* (residing in your Arduino sketchbook location) and open it in Notepad++.

```
program evalInput; // this is a JUSTINA program

function evalInput();
    var question, answer, flag;
    var amount = 0, totalAmount = 0;

    coutLine;
    while 1;
        // initialize 'answer' and 'flag'
        input question = "Enter an amount in metric ton (an expression is allowed)",
            answer = "", flag = NO_DEFAULT;
        if flag == CANCELED; break; end;

        trapErrors TRUE;
        totalAmount += amount = eval(answer) * 1000;
        trapErrors FALSE;
        if err() ; coutLine line(), "!!! Please enter a valid amount !!!", line();
        else; coutLine "amount entered = ", amount, " kg", line();end;
    end;

    coutLine line(), "*** total amount = ", totalAmount, " kg", line();
end;
```

Using a **while**...**end** structure and an **input** statement, this program repeatedly asks to enter an amount in metric tons and subsequently prints out this amount in kilograms. All amounts entered are summed up and when the user finally cancels the last input, the loop ends and the total amount entered is printed.

But the user can enter multiple amounts in one go, by entering an expression (like `2+3*4;`) when the program stops to request input. The *eval*(...) function will then parse and execute the expression the user entered.

But if the user makes an error in one of his entries, we don’t want the program to end execution with an error. We merely want to display an error message, indicating that the user entered an incorrect amount and let him try again.

We accomplish that by setting error trapping ‘on’ just before the *eval()* function, setting it ‘off’ again just after, and then testing for an error, using the *err()* function.

Now, load the program.

If we type this:

`evalInput();`	start the program
`2.3;`	enter an amount
`2 + 2.1;`	enter two amounts, using an expression
`"abc";`	an incorrect entry
`5;`	enter an amount
`\c;`	exit the program

The output will be

```
Justina> evalInput()

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 2300.00 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 4100.00 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton

!!! Please enter a valid amount !!!

===== Input (\c to cancel): =====
Please specify amount in metric ton
amount entered = 5000 kg

===== Input (\c to cancel): =====
Please specify amount in metric ton

*** total amount = 11400.00 kg

                                                   0
Justina>
```

2 + 2.1 metric ton

incorrect amount "abc"

total amount

If an error occurs and error trapping is not 'on' in the function where the error occurs, the function is ended and control returns to the calling function. If error trapping is 'on' in the calling function, the error can be trapped there (using the *err()* function to determine the nature of the error). If not 'on', the calling function is also ended and control passes to the caller of that function.

This goes on until a function with error trapping enabled is found in the call stack. The error can then be trapped in that function (note that an error can even be trapped in the command line).

An execution error will only be produced if no function in the call stack was found with error trapping 'on'.

13.6 Debugging

Stopping a program for debugging

A running program can be stopped for debugging in 4 ways: start a program in debug mode, insert **stop** commands in your program, set breakpoints or use Justina system callbacks.

Using system callbacks and a simple pushbutton, a program can be 'forced' to stop and enter debug mode (e.g., while in an endless loop). This will be discussed in Appendix D: 'Running background tasks: system callbacks'.

Setting breakpoints is by far the most powerful method: you don't need to alter and reload your program, you can set breakpoints anytime, not only before you start a program but also when it's currently stopped, you can specify breakpoint triggers and enter 'trace' expressions to view variable contents etc.

For now, let's start with the easiest ways to stop a program and see how we then can execute one statement at a time.

debug ;	If followed by a call to a program function; the program will enter debug mode, stopping the program before it executes its first statement. Use: command line only.
stop ;	Insert this statement where you want a program to stop, entering debug mode. The program will enter debug mode before it executes its next statement.
nop ;	no operation. Placeholder for **stop** (instead of removing **stop**, you could replace it by **nop**; this will not change program memory used nor will it change source line numbering).

Each time a program stops and enters debug mode, Justina will print two extra lines before the prompt: a 'STOP' line clearly signaling debug mode, and a line showing the next statement to execute, together with source line number and currently active function. These lines are printed to the 'debug out' stream.

- The debug out stream can be set to any stream, be it an IO device or an open SD card file (the latter is useful for logging debugging messages). See **setDebugOut** command, chapter 10: Input and output.
  On startup, the debug out stream is set to the same (default) IO device as the console.

Example

Locate file 'myFirst.jus' in folder *'libraries\Justina_interpreter\extras\Justina_language_examples'* (residing in your Arduino sketchbook location) and load the Justina program it contains.

```
program myFirstProgram; // this is a JUSTINA program

/*
    Very basic example of a Justina program. It demonstrates the use of local variables
    and executes a simple loop, printing a line to the console at each iteration.
    The function returns 6**2 (6 is the final value of the loop control variable)

    Function call: print5lines();
*/


function print5lines();      // this is a function

var i;                       // this is a LOCAL variable

for i = 1, 5;                // this is the start of a loop
    coutLine "line = ", i;   // this prints something
  end;                       // this is the end of a loop
  return i ** 2;             // this returns the square of i
end;                         // this the end of a function
```

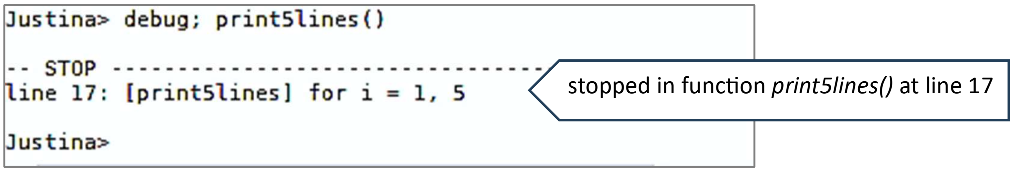
Now type:

`debug; print5lines();`

The program will immediately enter debug mode.

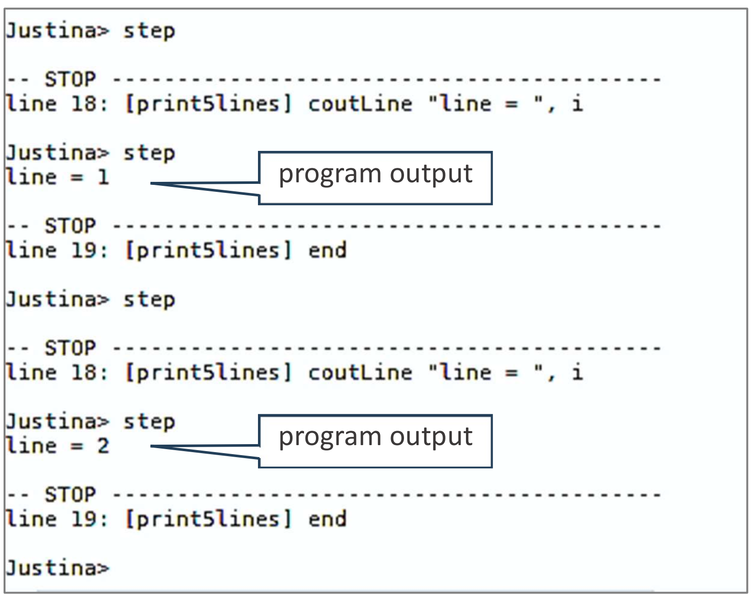
The console will first print a ‘STOP’ line to clearly indicate that the program has stopped and is in debug mode now.

On the next line it will print the line number of the *next* statement to execute, the function (between square brackets) and the source statement ("**for** I = 1, 5", which you can verify in Notepad++).



Stepping through a program

Now, enter command **step** a few times. Each time, Justina executes one statement and prints out the statement to be executed next.



As you have seen in the previous example, when the system enters debug mode, the command line is ready to accept input again. But because a program is stopped, we call this the ‘*debug command line*’, because a number of debugging commands become available.

Command **step** is part of a series of commands to execute one, or a few, program statements, staying in debug mode. There is also a command to exit debug mode and resume execution.

Note that these commands will produce an error if no program is stopped in debug mode.

step ;	Execute one statement and enter debug mode again before the next statement is executed. If a statement contains a call to another function, Justina will step into that function and stop there. Use: command line only.
loop ;	If currently stopped inside a control structure, executes statements until the **end** statement is reached: Justina will execute one iteration of a loop (**for**...**end**; ...) or the statements within an **if**...**end** structure, and will stop and enter debug mode again before the control structure **end** statement, making this the next statement. If not currently within a control structure, then **loop** behaves like **step**. Use: command line only.
bStepOut ;	'block step out' - command line only. If currently stopped inside a control structure, continue execution until all statements of the control structure have been executed. The program will stop and enter debug mode again at the first statement <u>after</u> the control structure **end** statement. If not currently within a control structure, then **bStepOut** behaves like **step**. Use: command line only.
stepOut ;	Continues execution until all statements of the current function have been executed. The program will stop and enter debug mode again in the calling function, <u>after</u> the statement with the function call. If the function was called directly from the command line, then execution will continue there (Justina stops only if within a program). Use: command line only.
stepOver ;	Execute one statement and enter debug mode again before the next statement is executed. If a statement contains a call to another function, Justina will <u>not</u> step into that function (it will not stop and enter debug mode there). Use: command line only.
go ;	continue execution again when a program was stopped in debug mode. use: command line only.

Sometimes it is useful to manually skip execution of part of a program that is being debugged, because that part is not relevant in the context of debugging. This is accomplished by the **setNextLine** command.

setNextLine line ;	Change the 'next statement' (where execution will resume) to the (first) statement starting on the indicated source line. The statement must be part of the function where the program is currently stopped and it is not possible to move control <u>into</u> a currently uninitialized block structure (a loop or **if**...**end** structure). You can always move control out of a block structure, however. Note that this command will produce an error if no program is stopped in debug mode. Use: command line only.

When a program is stopped in debug mode, the user has access to the stopped function's local variables from the command line: a user can examine variables and even change their values, in the same way he accesses user variables or global program variables.

But if multiple variables (or constants) with the same name exist, the parser will first look for a user variable with that name, then for a global program variable with that name and, only if none of these two exist, for a local variable of the stopped function. So, we need a way to tell the parser to look for that function's local variable immediately.

#	This is a prefix (not an operator), optionally placed in front of a variable name, to force the parser to interpret the variable as a local variable of a stopped function in case a user variable or a global program variable with the same name would exist as well.

Notes:

- As soon as one of the above commands executes, the parsed command line is deleted and command line execution will terminate: for instance, if you type `1 + 2; step; 3 + 4;` Justina will execute a statement from the stopped program but the remaining expression `3 + 4;` will never get executed – which is quite logical, because there's no way control could still return to there.
- Instead, If a stopped program finally ends (terminates), the original command line (containing the call to the program) will continue execution: although the original command line text in the serial monitor or terminal may have been overwritten by debugging commands, it's parsed statements were saved and execution will continue there, just as if the program was executed without debugging – which is as it should be.

Aborting a program

A program stopped in debug mode can be aborted by using the **abort** command.

abort ;	Terminate a currently stopped program, releasing all memory it occupied for local function variables. This doesn't influence global program variables (as they are created during program parsing) or user variables. Use: command line only.

Note that a user can also abort running code by using system callbacks. See Appendix D: '*Running background tasks: system callbacks*'.

13.7 Tracing variables and expressions

Tracing provides a way to automatically review the contents of variables, and even the result of expressions, during debugging.

To inform Justina about the variables or expressions you would like to review, using the **trace** command. During debugging, you'll then see the evolution of selected variable contents or expression results.

trace traceString ;	This command stores a 'traceString': a list of expressions, stored as a single string. The expressions within the string are **separated by semicolons**. A trace string is used to automatically 'trace' the values of specific variables - including the stopped function's local variables - while you execute statements during debugging: the expressions stored in the trace string will be automatically parsed and evaluated, and the results printed, each time control returns to the command line while in debug mode. Tracing is not active if 'traceString' is set to an empty string ("") or a program is not in debug mode. The string <u>cannot</u> contain command statements, *eval()* functions and calls to program functions. No other restrictions apply: you may use variables and constants, operators, call built-in functions and 'external' functions you write in c++. To include a string constant within 'string', use escape sequences (see chapter 4: *Data types*), or use the *quote()* function.
viewExprOn ;	While tracing, precede each value traced by the corresponding expression text and a colon
viewExprOff ;	While tracing, print values traced only - without the corresponding expression text

Traced variable values and expression results are printed to the 'debug out' stream and will appear in a separate 'TRACE', line in between the 'STOP' line and the line showing the next statement.

- On startup, the debug out stream is set to the IO device also designated as default console. If this is not wanted, the debug out stream can be set to any stream, be it another IO device or an open SD card file (the latter is useful for logging debugging and tracing messages).

The 'TRACE' line starts with a <TRACE> label, followed by a comma-separated list of values. Depending on the current setting (see table above), each value may be preceded by the corresponding expression text and a colon.

If parsing and evaluation of an expression in the trace string produces an error:

- parsing error: 'ErrP' followed by the <u>parsing error</u> number is printed instead of the respective expression and value (even if viewing expressions is currently Off)
- evaluation error; the respective expression is printed, followed by a colon and 'ErrE' plus the <u>execution error</u> number

<u>Example: program 'factorial'</u>

We discussed this program earlier in this chapter. Let's now execute it step by step, while reviewing the contents of local variable 'n' and 'fact_n' in function *fact()*.

Locate file 'fact.jus' in folder *'libraries\Justina_interpreter\extras\Justina_language_examples\', residing in your Arduino sketchbook location*) and load the Justina program it contains.

Open the file in notepad++ as well, to be able to follow execution of the program.

First, set the trace string: `trace "n; fact_n";`

While tracing, view expressions (text) as well, not only values: **viewExprOn**

Now let's calculate the factorial of 3 (3!) and trace the evolution of the variables.

Start debugging: `debug; fact(3);`

Then step through the program until the result, 6, is printed.

Remember that the **function** and **var** commands are non-executable statements; these statements are skipped during execution.

```
Justina> trace "n; fact_n"
Justina> debug; fact(3)                    calculate 3!

-- STOP -----------------------------------------
<TRACE> n: 3, fact_n: 0
line 21: [fact] if (n > 2)

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 3, fact_n: 0
line 22: [fact] fact_n = n * fact(n - 1)   calculate 3 x 2!

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 2, fact_n: 0
line 21: [fact] if (n > 2)

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 2, fact_n: 0
line 23: [fact] else

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 2, fact_n: 0
line 24: [fact] fact_n = n

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 2, fact_n: 2
line 25: [fact] end

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 2, fact_n: 2
line 27: [fact] return fact_n              return 2!

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 3, fact_n: 6
line 25: [fact] end

Justina> step

-- STOP -----------------------------------------
<TRACE> n: 3, fact_n: 6
line 27: [fact] return fact_n              return 3!

Justina> step
                                                 6
Justina>
```

Step-by-step execution of function fact(3)

```
function fact(n);
    var fact_n = 0;

    n = cInt(n);

    if (n > 2);
        fact_n =  n * fact(n-1);
    else;
        fact_n = n;
    end;

    return fact_n;
end;
```

Excerpt of source file 'fact.jus'

Each time Justina stops in debug mode, the console will now print an extra line: it starts with label "<TRACE>" followed by all variables (or even expressions) in the trace string with their value.

If a global program variable or a user variable exists with the same variable name as the local variable, precede the variable name with the '#' character to force the parser to select the stopped function's local variable.

Example: `trace #n`.

Printing the call stack

During debugging, sometimes it helps to ‘see’ how deep functions calling each other are currently nested. We can visualize this by printing the ‘call stack’. This is especially useful when dealing with recursive function calls.

listCallStack [streamNumber] ;	This command prints the current call stack. If a program is stopped for debugging, this shows us what function was initially called from the command line, and the tree of functions called when the program was stopped. The function that was executing instructions (the 'deepest' instruction in the call stack) is shown first. The ‘streamNumber’ argument can refer to any available external IO device **and** to any open file. If no device number is specified, prints to console.

Example

Referring to the previous example, let’s now calculate the factorial of 5.

Start debugging: `debug; fact(5);`

Execute '`step;`' 5 times.

Then execute command `listCallStack CONSOLE;` (the argument is optional; CONSOLE is the default)

```
Justina> listCallStack CONSOLE

fact()
|__ fact()
    |__ fact()
        |__ command line


-- STOP ------------------------------------------
<TRACE> n: 3, fact_n: 0
line 22: [fact] fact_n = n * fact(n - 1)

Justina>
```

This shows that control is currently 3 levels deep in function ‘fact’ and the next line to execute is line 22.

13.8 Breakpoints

Breakpoints allow you to 'mark' specific program statements where you want the program to stop and enter debug mode. Breakpoints are not inserted in your program; they are maintained separately in a breakpoint table and they don't change your program in any way. You can enter a maximum of 10 breakpoints.

Breakpoints are extremely helpful while debugging a program. You can

- enter and change breakpoints dynamically
- add a separate trace string for each breakpoint, specifying multiple variables or expressions to be traced
- add a separate trigger (optional) for each breakpoint, specifying a condition for stopping the program

Breakpoints are set using the **setBP** commands, which has two forms

setBP line [, line, ...] ; - or - **setBP** line, traceString [, trigger] ;	Set breakpoints for specific source lines, forcing the program to stop if the statement starting on one of these source lines is reached while the program was running. This does not change previously set breakpoint attributes (see next). - or - Set a breakpoint for a source line, forcing the program to stop if the first statement starting on that source line is reached while the program is running. At the same time, this second form stores a trace string (same format and restrictions as **trace** command), but only applicable to this breakpoint. If 'traceString' is set to an empty string (""), no variables (or expressions) are traced when this breakpoint is hit. If the optional trigger is specified, it's stored as well. The trigger can either be a hit count or a condition: • condition: a string, containing one expression. Each time the source line is reached, the expression is parsed and evaluated. If the result is numeric and not equal to zero (TRUE), the program will stop and enter debug mode. Otherwise, execution will continue. • hit count: a number indicating the number of times this source line must be reached before the program stops. An internal counter keeps track of this. This counter is reset each time the breakpoint is hit and when the **setBP** statement sets a new hit count. Use: command line only.

Each time a program stops and enters debug mode because a breakpoint was encountered, Justina will print a 'BREAK' line instead of a 'STOP' line to indicate that a breakpoint was hit, and (as with a normal stop) a line showing the next statement to execute. Output is printed to the 'debug out' stream.

Traced variable values and expression results are printed to the 'debug out' stream as well, on a separate line, starting with the label <BP TR> (breakpoint trace) and in between the 'STOP' line and the line showing the next statement.

- The debug out stream can be set to any stream, be it an IO device or an open SD card file (the latter is useful for logging debugging messages). See **setDebugOut** command, chapter 10: Input and output.
  On startup, the debug out stream is set to the same (default) IO device as the console.

Notes

- **viewExprOn** and **viewExprOff** commands affect printing of expressions during BP tracing as well.

- Errors during parsing and evaluation of trace string expressions when a breakpoint is hit, are reported in the same way as when parsing global trace string expressions (see **trace** command).
- An error during parsing and evaluation of a *trigger* condition (a string expression), as well as a non-numeric result is interpreted as a FALSE condition: the program will not stop at the respective breakpoint.
- When a new program is loaded or a program is cleared, all breakpoints set are deleted.

Example: program 'factorial'

We already executed this program step by step, to illustrate how debugging and tracing works.

Now we will 'debug' this program again, but by using breakpoints.

Load program 'fact.jus' again (**loadProg** command). And, again, open it in Notepad++ as well, to be able to follow where control is during debugging.

It's important to place breakpoints 'strategically', in order to have a good understanding of what the program does, based on the contents of variables used in the program ('n' and 'fact_n').

- we'll place breakpoints at the two lines containing an expression: these are lines 22 and 24. Here, we're interested in the value of variable 'n' (variable 'fact_n' is zero at this point: 'fact_n' is a local variable, it has just been initialized – see line 19)
- the **return** statement returns the result of these expressions, so here we'll put a breakpoint to trace variable 'fact_n'.

Set the breakpoints now. Enter these lines:

`setBP 22, "n";`	argument of the active 'fact' function (for n > 2)
`setBP 24, "n";`	argument of the active 'fact' function (for n <= 2)
`setBP 27, "fact_n";`	return value of the active 'fact' function

We'll not set a trigger for these breakpoints at this time.

While tracing, view expressions (text) as well, not only values: **viewExprOn**

Now let's calculate the factorial of 3 (3!) again and trace the evolution of the variables.

Start the program: `fact(3);`

Then continue execution, using **go** instead of **step**, and do that until the program ends and the result, 6, is printed.

➢ You don't need to start the program in debug mode, nor do you need to insert **STOP** statements: you have set breakpoints instead.

Now, the program only stops where you want it to stop – and without inserting **stop** commands that need to be removed again afterwards.

```
Justina> fact(3)                          calculate 3!

-- BREAK ------------------------------------------
<BP TR> n: 3
line 22: [fact] fact_n = n * fact(n - 1)   calculate 3 x 2!

Justina> go

-- BREAK ------------------------------------------
<BP TR> n: 2
line 24: [fact] fact_n = n                 calculate 2!

Justina> go

-- BREAK ------------------------------------------
<BP TR> fact_n: 2
line 27: [fact] return fact_n              return 2!

Justina> go

-- BREAK ------------------------------------------
<BP TR> fact_n: 6
line 27: [fact] return fact_n              return 3!

Justina> go
                                                6
Justina>
```

Executing function fact(3) with breakpoints set

```
function fact(n);
    var fact_n = 0;

    n = cInt(n);

    if (n > 2);
        fact_n =  n * fact(n-1);
    else;
        fact_n = n;
    end;

    return fact_n;
end;
```

Excerpt of source file 'fact.jus'

Other breakpoint commands:

clearBP line [, line, ...] ;	Clear breakpoints for specific source lines and clear associated breakpoint trace strings and conditions if set. Use: command line only.
enableBP line [, line, ...] ;	Enable breakpoints for specific source lines. This requires that the respective breakpoints are set. If not, this command will produce an error. Note: when a breakpoint is initially set, it is enabled by default. Use: command line only.
disableBP line [, line, ...] ;	Disable breakpoints for specific source lines. These breakpoints will be inactive until they are enabled again. This command requires that the respective breakpoints are set. If not, this command will produce an error. Use: command line only.
BPon ;	Enable breakpoints. This is a global setting and does not influence any of the defined breakpoint settings. This is the default status. Use: command line only.
BPoff ;	Disable breakpoints. This is a global setting and does not influence any of the defined breakpoint settings. Use: command line only.

To get an overview of all current breakpoints, use the **listBP** command.

listBP [streamNumber] ;	Print a list of all currently defined breakpoints with their attributes. The list is sorted by source line number. The 'device number' argument can refer to any available external IO device **and** to any open file. If no device number is specified, prints to console.

Referring to the previous example, this is the output of the **listBP** command:

```
Justina> listBP CONSOLE

Breakpoints are currently ON

source   enabled   view &
  line             trigger
------   -------   -------
    22      x      view : n;
                   always trigger
    24      x      view : n;
                   always trigger
    27      x      view : fact_n;
                   always trigger

Justina>
```

Now, let's adapt the trace string for line 22, to display not only the value of 'n', but also a random number between 0 and 999. In addition, we'll disable the breakpoint for line 24, set a condition for line 22 and a hitcount for line 27.
To make sure that the parser selects local variable 'n' and not a global or user variable with the same name , we'll use '#n' in trace and trigger expressions.

`setBP 22, "#n; random(1000)" , "#n<=4"`	break when n is less than or equal to 4
`setBP 27, "fact_n", 3`	break every three times the line gets executed
`disableBP 24`	

Calculate the factorial of a few numbers and see what happens.

13.9 Executing a program while one or more programs are stopped

While a program is stopped for debugging, you can start another program ‘instance’ . You can even stop that second running program as well, debug it, trace its variables etc. But you cannot switch to a previously stopped program and continue execution there before all newer program ‘instances’ were ended (or aborted). If more than one program is stopped, Justina will indicate that in the 'STOP' line while debugging.

Note: **only one program file** can be loaded and parsed in program memory at any one time. So, starting a new program ‘instance’ means calling one of the functions available in the parsed program and executing it – possibly the same function that started the currently stopped program(s). The only thing to take into account is that global program variables and user variables are shared.

From inside a running program, you can access the local variables of the function where the last program instance was stopped. To do this from inside a running program, use the *eval()* function and add prefix ‘#’ to the stopped function’s variable names (same prefix as you would use from the command line or from within a trace string).

Example: if the stopped function has a local variable ‘count’, then you could do this from inside a running program:

`eval(" #count ");`	to return the value of variable ‘count’
`eval(" #count += 7 ");`	to add 7 to ‘count’ and return the new total

Printing the call stack (**listCallStack** command) will print a separate function tree for each stopped program.

14 Appendices

Appendix A Creating a Justina object and choosing startup options

Creating a Justina_interpreter object using default values

The simplest way to create a Justina object is by using this statement:

```
Justina justina;
```

This sets Serial as the single IO 'channel' available for Justina and assumes that the Arduino Serial Monitor (or another serial terminal program or device) will act as console. Moreover, if an SD card board is connected, Justina is allowed to access SD cards to create, read and write files, etc. The SD card chip select pin is set to pin 10 by default.

Creating a Justina object, specifying an SD card mode and chip select pin

```
Justina justina(cardMode [, CSpin]);
```

cardMode: use one of the following public Justina constants (precede by ' Justina:: '):

`Justina::SD_notAllowed`	card reader not present or card operations not allowed (maybe, SDcard is in use by the calling Arduino program)
`Justina::SD_allowed`	card reader is allowed but SD card will not be started (mounted) automatically (maybe no SD card is inserted). **This is the default**
`Justina::SD_init`	start (mount) SD card when calling Justina begin*()*
`Justina::SD_runStart`	when calling Justina begin*()*: start (mount) SD card, load Justina program "SD_start.jus" from SD card if present and execute user function start*()* if found

CSpin: SD card chip select pin (optional). Connect this Arduino pin to the SD card reader Chip Select pin. The default is Arduino pin 10.

Creating a Justina object connecting to multiple IO devices

Justina can handle up to 4 input and output devices, represented by Stream objects and Print objects, respectively.

```
Justina justina(inputs, outputs, count [, cardMode , CSpin]);
```

inputs: a *Stream ** array with pointers to input streams (Serial, a TCP IP client, an ASCII keyboard...)
outputs: a *Print ** array with pointers to output streams (Serial, a TCP IP client, an OLED or LCD screen, ...)
count: the number of inputs and outputs defined (minimum 1, maximum 4)
SDcardMode: as described above (optional)
CSpin: as described above (optional)

Example

For example, if you're using a Serial monitor and an lcd display (number of devices = 2):

```
Stream* pExtInput[2]{ &Serial, nullpr};
Print* pExtOutput[2]{ &Serial, &lcd};
Justina justina(pExtInput, pExtInput, 2);
```

(assuming an 'lcd' object has been created - see for instance the Arduino IDE example ' LiquidCrystal ')

- ☞ If an output device has no corresponding input device (e.g., the lcd display in this example), enter a 'nullptr' in the corresponding position within the input stream array. The same logic applies if an input device has no corresponding output device.
- ☞ Justina uses the input and output devices referenced in the first array position as default console input and output, respectively. None of these can be a nullptr.
- ☞ Typically, Serial will be entered in the first array position (default console). However, any capable IO device can be set as default console.

Example

```
Stream* pExtInput[1]{ &Serial };
Print* pExtOutput[1]{ & lcd};
Justina justina(pExtInput, pExtInput, 1);
```

In this example (assuming an 'lcd' object has been created), the Arduino Serial monitor will be used as console input only; console output will be sent to an LCD display (which is probably not very useful).

Example programs

The Justina library contains two sketches demonstrating the use of additional IO devices, next to Serial.

- Justina_OLED.ino adding OLED displays as extra Justina output devices
- Justina_TCPIP.ino adding a TCP IP terminal as extra Justina output device

Arduino IDE: *File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_OLED*
File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_TCPIP

Appendix B Changing the size of memory allocated to Justina

By default, Justina sets the size of specific memory areas, taking into account the available RAM of the Arduino board used.

	maximum allowable	Arduino RP2040 and Arduino ESP32 boards	Arduino SAMD boards
program memory size in bytes (*)	65536	65536	4000
max. program variable names (**)	255	255	64
max. user variables	255	255	64
max. static variables	255	255	32
max. program functions	255	255	32

(*): Minimum is 2000 bytes. This includes 500 bytes of program memory for parsed immediate mode (user) commands, leaving 1500 bytes (which is sufficient for a tiny program).

(**): Program variable names are shared between global, local and static program variables: names are stored only once (but a global, a local and a static program variable using the same variable name are all distinct variables, of course).

Depending on your specific requirements, these values can be increased or decreased by editing a specific 'constants.h' file, but this WITHOUT CHANGING ANY OF THE FILES IN THE JUSTINA LIBRARY. Changes you make in a library file would be overwritten each time the library is updated.

The constants.h file

- Locate folder *'libraries\Justina_interpreter\extras\Justina_constants'* in your sketchbook location.
- Copy this folder (*'Justina_constants'*) to folder *'libraries'*. You have now a library folder *'Justina_constants'* within the *'libraries'* folder, next to the *'Justina_interpreter'* library folder.

Change the size of specific memory areas

- Open file *'libraries\Justina_constants\Justina_constants.h' for editing*
- Change the values next to the preprocessor #DEFINE directives. For example:

```
#define PROGMEM_SIZE 2000
#define MAXVAR_USER 100
#define MAXVAR_PROG 100
#define MAXVAR_STAT 100
#define MAXFUNC 50
```

- Save the file

☞ if you don't want to change the default for a specific value (as in the table above), comment out the respective line (' //').

Appendix C Example programs

c++ examples

The Justina interpreter library contains a number of c++ examples, which can be selected from the *Arduino IDE menu:*

File -> Examples -> Examples from custom libraries -> Justina interpreter -> (select an example from the list)

Each of these sketches will start Justina, but (apart from the first sketch, which is basic) they demonstrate specific features built-in into Justina (e.g., adding extra IO channels to Justina, next to the console).

Example	Scenario
Justina_easy	As the name suggests, this example is straightforward. It demonstrates how to start Justina. When the program is executed, you should see the Justina prompt appearing on the console. You can then start typing user commands in the command line, load and execute Justina programs, etc. Hardware required: none.
Justina_systemCallback	Demonstrates the use of system callbacks to detect Justina stop, abort, console reset and kill requests, retrieve the current Justina status (e.g., to signal that a user error occurred) and blink a heartbeat led. Hardware required: 5 LEDs, 2 pushbuttons, resistors. More information: Appendix D: '*Running background tasks: system callbacks*'.
Justina_userCPP	Demonstrates how to extend Justina functionality by writing 'user c++' functions. You can then call these user c++ functions from the Justina command line, just like any other Justina function, with the same syntax, using an alias as function name and passing scalar or array variables as arguments. Hardware required: none. More information: Appendix E: *calling user c++ functions.*
Justina_userCPP_lib	Demonstrates how to create a Justina 'user c++ library' file. It also shows how to pass arrays (by reference) to a user c++ function. Hardware required: none. More information: Appendix E: *calling user c++ functions.*
Justina_OLED	Demonstrates how to set up OLED displays as additional Justina output devices, next to Serial. You can then print data to the OLED displays in the same way you print to the Justina console. You can change the console (in this case, for output only) to an OLED display. Hardware required: OLED display with SH1106 controller communicating over SW SPI and/or OLED display with SSD1306 controller communicating over I2C. The sketch sets up the OLED displays as additional output devices IO2 and / or IO3, next to Serial (IO1, CONSOLE).

Justina_TCPIP	This is probably the most 'complex' of the examples provided. The program demonstrates the setup needed for various Justina features, namely • setting up Arduino as a TCP/IP server in order to use a TCP/IP client terminal as an additional output device, or to build a Justina HTTP server on top of it (see Justina language examples, below). • using Justina *system callback functions* to maintain the TCP/IP connection, to blink a heartbeat LED and to set status LEDs indicating the TCP/IP connection state. • using Justina user c++ functions (user *callback functions*) to let Justina control the TCP/IP connection. Hardware required: 4 LEDs, 4 x 220 Ohm resistor. More information: - next paragraph ('*The Justina_TCPIP sketch in some more detail*') - Appendix D: '*Running background tasks: system callbacks*' - Appendix E: *calling user c++ functions* - Appendix G: *Installing YAT terminal*

The 'Justina_TCPIP' sketch in some more detail

The sketch sets up a TCP/IP server communicating over a second IO channel (IO2), next to Serial (IO1, CONSOLE).

Before running this sketch, you'll have to prepare a couple of things:

First, enter the data that you need to keep private in the secrets.h file:

```
#define SERVER_SSID "mySSID"
#define SERVER_PASS "myPassword"
```

Also set the static IP address for the server (your Arduino), and set gateway address, subnet mask and DNS address to correspond to your local network settings. Also enter the server port. Example:

```
const IPAddress serverAddress(192, 168, 1, 45);
const IPAddress gatewayAddress(192, 168, 1, 254);
const IPAddress subnetMask(255, 255, 255, 0);
const IPAddress DNSaddress(8,8,8,8);
const int serverPort = 8085;
```

As the server address is static, it won't change over time, which makes it easier for clients to connect.

In your router settings:

(1) set the static IP address for your Arduino (same that you entered in secrets.h)
(2) if you want access from outside your LAN: enable port forwarding

> ⚠ If not familiar with this topic, it is suggested that you study and run a few of the standard Arduino WiFi examples available in the Arduino IDE first.

In the sketch, 4 Arduino pins are defined as outputs. Connect each output pin to the anode of a LED (refer to the sketch for the output pin numbers) and connect each cathode to one terminal of a resistor. Wire the other terminal to ground.

LEDs connected to pins:

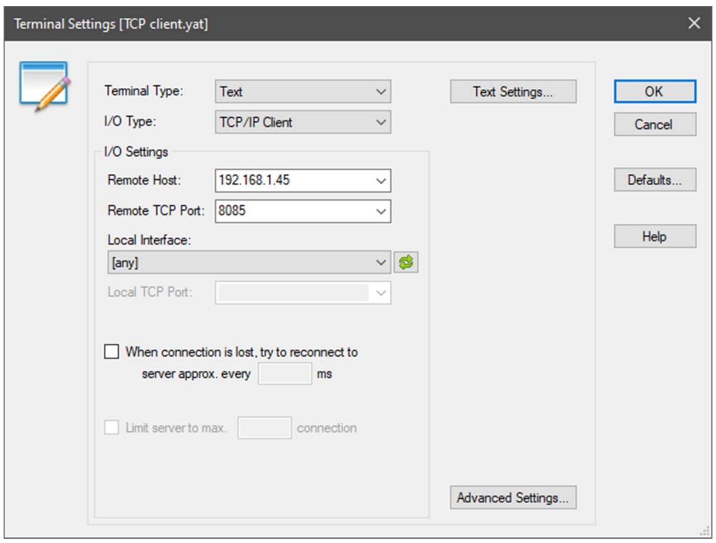
HEARTBEAT_PIN	when blinking, signals that your sketch is running
DATA_IO_PIN	blinks while Justina is sending or receiving data to/from any defined IO device
WiFi_CONNECTED_PIN	ON when WiFi is connected
TCP_CONNECTED_PIN	blinking when waiting for a TCP/IP client, ON when a TCP/IP client is connected

Testing your sketch

To test proper operation of the TCP/IP server, you'll need a TCP/IP client to connect to it. This TCP/IP client can then read and write data from/to Justina (you could even change the console to the TCP/IP client).

A convenient way to setup a TCP/IP client, is to use YAT (see Appendix G: *Installing YAT terminal*).

To configure YAT as a TCP client, follow the steps in Appendix G: *Installing YAT terminal,* but select TCP/IP as IO type, fill in the static server address ('Remote Host') and port (as setup in your sketch) and deselect the check boxes beneath.



Configuring YAT as TCP/IP client

- If you already use YAT as the Justina console: simply open a second YAT instance on your computer and configure it as a TCP client
- you could also set up the TCP/IP terminal on another PC - or even from outside your local network, but you'll need to enter the external (WAN) IP address and port then.

Connect the TCP/IP terminal (supposing you use YAT: click the green 'Start Terminal' button).

You will now have two terminal windows open: a Serial terminal and a TCP/IP terminal.

Start Justina (type "j" (+ ENTER), as requested by the Serial Terminal).

Now enter `printLine IO2, "hello";` . If all is well, the text is printed on the TCP/IP terminal window.

To test sending text to the TCP/IP client, we'll make a small 'immediate mode program'. Enter these 2 lines:

```
var s;
while 1; s=readLine(IO2); if strCmp(s,"end"+line())==0; break;...
                               ...elseif strCmp(s, "")!=0; cout s; end; end;
```

This 'program' continuously waits for, and prints, data from the TCP/IP terminal. It quits if the text 'end' is received.

Try it; return to the Justina prompt by sending "end" to Justina.

User c++ functions to let Justina control the TCP connection

`cpp_WiFiOff();`	disconnect from the WiFi network
`cpp_WiFiRestart();`	reconnect to the WiFi network
`cpp_TCPoff();`	stop the connection with a client and do not connect to a new client
`cpp_TCPon();`	wait for a client to connect
`cpp_stopClient();`	stop the connection with the client, keep waiting for a new client
`cpp_setVerbose(verbose);`	verbose = TRUE: print debug messages to Serial when connection status changes. FALSE: do not print debug messages
`cpp_localIP(s);`	get local IP address. The static address is returned as a string in a variable 's', which must be initialized with a string of at least 15 characters): `cpp_remoteIP(s=space(15));`
`cpp_remoteIP(s);`	get the client IP address. The address is returned as a string in a variable 's', which must be initialized with a string of at least 15 characters): `cpp_remoteIP(s=space(15));`
`cpp_connState();`	returns the current connection state: 0: WiFi not connected 1: trying to connect WiFi 2: WiFi connected - TCP/IP disabled 3: WiFi connected - waiting for TCP/IP client 4: WiFi connected - TCP/IP client connected

Turn your Arduino into a simple web server (HTTP server)

The Justina_interpreter library contains two Justina language examples transforming your Arduino into a webserver (HTTP server). Check out the next paragraph for details.

Justina language examples

The Justina interpreter library contains a number of Justina language examples, stored in repository folder

(Arduino sketchbook location) \ libraries\Justina_interpreter\extras\Justina_language_examples

File names obey the 8.3 file name format, to make them compatible with the Arduino SD card file system. Also, these files have '.jus' as extension: opening these files in Notepad++ will automatically invoke Justina language highlighting (if the Justina language extension is installed in notepad++ , see Appendix F: *Installing Notepad++ and the Justina language extension*).

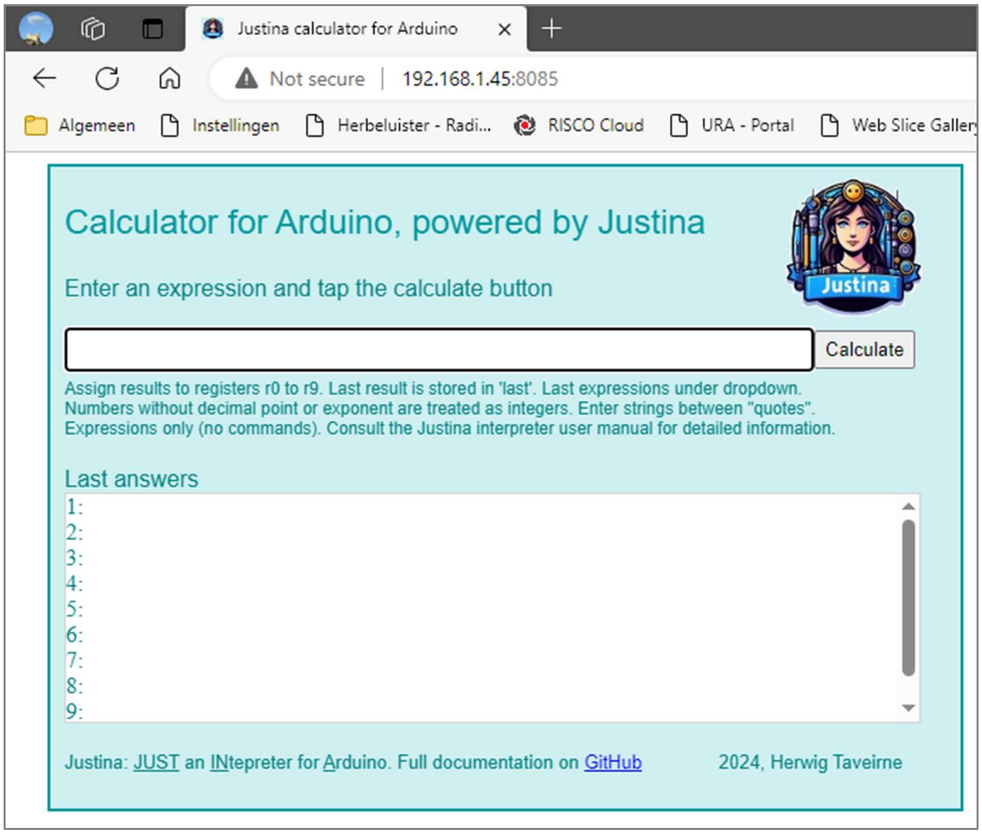
The example files are:

myFirst.jus	a really simple Justina program, printing a few lines of text on the console
fact.jus	a recursive method to calculate factorials
input.jus	ask for user input; parse and execute that input within a running program
overlap.jus	two methods to print lines with overlapping print fields
start.jus	program to set the display mode, display width, floating point print format, integer print format and angle mode. This program will automatically run function '*start()*' (without arguments) right after Justina is started, if: • your Arduino is equipped with an SD card • this file ('start.jus') is stored in folder 'Justina' (path '/Justina/start.jus'). • Justina startup options must allow autostart (see Appendix A: *Creating a Justina object and choosing startup options*)
SD_test.jus	perform some basic SD card tests
SD_parse.jus	write formatted data to an SD card, read it back and immediately parse this data into variables
web_swit.jus	a simple HTTP server, drawing a webpage with 5 'on/off' buttons, and maintaining the state of 5 'switches'. Buttons representing a switch that is currently 'on' are drawn in a red color. Clicking a switch acts like a toggle. For this program to work, first compile and load Arduino c++ program 'Justina_TCPIP', which sets up your Arduino as a TCP server, and test that the TCP server works as expected (see above). Justina program 'web_calc.jus' adds an additional layer on top, turning your Arduino into a HTTP server (web server). To test, open a web browser and type the complete url, starting with http://... `http://nnn.nnn.nnn.nnn:port` (fill in IP address and port)
web_calc.jus	a web server, creating a web page functioning as a (working) scientific calculator with 10 user registers and a 'last result' register. An input box allows you to type in any expression (no commands) as long as it adheres to the Justina syntax: internal Justina functions, Justina user functions and Justina user cpp functions are all allowed. When submitted, the expression is evaluated by Justina and the result (or the error, if an error is produced) is returned. The 10 last results are shown in an answer box below the input box ('Last answers'). For this program to work, first compile and load Arduino c++ program 'Justina_TCPIP', which sets up your Arduino as a TCP server, and test that the TCP server works as expected (see above). Justina program 'web_calc.jus' adds an additional layer on top, turning your Arduino into a HTTP server (web server).

To test (or use !) the scientific calculator, open a web browser and type the complete url, starting with http://...

`http://nnn.nnn.nnn.nnn:port` (fill in IP address and port)

This is what you should see:



Notes:

- Integer and floating-point results are displayed according to the currently set display format for integers resp. floating point numbers (two user functions are provided to change integer and floating-point number formatting, respectively).
- Use the standard Justina fmt(...) function to override the set display format
- String results are displayed with any control characters replaced by a small white box
- This example uses the SD card: it expects to find a Justina logo file and icon file (jus_logo.jpg and jus_icon.jpg) in SD card directory "/justina/images". NOTE: if an SD card is not connected or the image files are not found, the web page will still be displayed (the calculator will still work), but without logo and icon.

Appendix D Running background tasks: system callbacks

The purpose of system callbacks (executed in the background, multiple times per second), is to
- ensure that procedures that need to be executed at regular intervals (e.g., maintaining a TCP connection, etc.) continue to be executed while control is within Justina
- detect stop, abort, console reset and kill requests (e.g., to request aborting a running Justina program stuck in an endless loop), when a user presses a pushbutton wired to an input pin
- retrieve the Justina interpreter state (idle, parsing, executing, stopped in debug mode, error), for instance to blink a led or produce a beep when a user error is made

This eliminates the need for Justina to have any knowledge about the hardware (pins, ...).

If enabled, the system callback function is called:
- whenever Justina is idle (waiting for input): constantly
- when busy (parsing or executing): after a complete statement is parsed or executed, provided that 100 milliseconds have passed since the previous call

System callback functions should be kept short (handled like interrupt service routines) in order not to slow down Justina operation.

The callback function communicates with Justina via a set of 32 application flags, some used to pass the Justina status to the callback function and some to read back 'requests' provided by the callback function. Most of the flags are unassigned.

Justina provides a list of public long constants, all starting with prefix 'appFMT_', that can be used to test, set or clear application flags in the c++ callback functions.

Status info provided by Justina to the callback procedure

2 application flags pass the current Justina state to the callback procedure:

`Justina::appFMT_statusMask`	use this mask before testing Justina status (2 bits)
`Justina::appFMT_idle`	Justina is idle
`Justina::appFMT_parsing`	Justina is currently parsing a program or a user command
`Justina::appFMT_executing`	Justina is currently executing a program or a user command
`Justina::appFMT_stoppedInDebug`	Justina is currently stopped in debug mode
`Justina::appFMT_statusAbit`	status mask bit 0
`Justina::appFMT_statusBbit`	status mask bit 1

One application flag informs the callback procedure that an error has occurred and 1 flag is set if since the last call to the callback procedure, Justina sent or received data to / from an external IO device.

`Justina::appFMT_errorConditionBit`	set if an error has occurred, reset otherwise
`Justina::appFMT_dataInOut`	currently sending or receiving data

Requests provided by the callback procedure to Justina

The callback procedure can set 4 individual bits to request a specific Justina action:

`Justina::appFMT_consoleRequestBit`	request to reset Justina console to the default
`Justina::appFMT_killRequestBit`	request to kill (exit) Justina
`Justina::appFMT_stopRequestBit`	request to stop a running Justina program in debug mode
`Justina::appFMT_abortRequestBit`	request to abort running Justina code

A flag needs to be set only once (during a single call to the callback procedure) to trigger the requested action.

Example programs

The Justina library contains 2 sketches that make use of system callbacks.

- Justina_systemCallback.ino demonstrates how to use system callbacks to provide a visual indication of the current interpreter state (idle, executing, error, ...)
- Justina_TCPIP.ino demonstrates how to use system callbacks to maintain a TCP connection

Arduino IDE: *File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_systemCallback*
File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_TCPIP

Appendix E calling user c++ functions

Built-in Justina functionality can be extended by writing specific functions in c++. Such functions may include time-critical user routines, functions targeting specific hardware, functions extending functionality in a specific domain, etc. These functions must then be 'registered' with Justina and given a 'Justina function name' (an alias).

From then onward, these C++ functions can be *called just like any other Justina function*, with the same syntax, using the alias as function name and passing scalar or array variables as arguments.

The steps involved are detailed below. But the Justina library contains 3 sketches containing examples of user c++ functions, which you can call from Justina (even from the command line):

- Justina_userCPP.ino some examples of user c++ functions
- Justina_userCPPlib.ino demonstrates collecting user cpp functions in a 'user c++ library' file
- Justina_TCPIP.ino user cpp functions to set and get TCP/IP attributes

Arduino IDE: *File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_userCPP*
File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_userCPPlib
File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_TCPIP

The second sketch (Justina_*userCPPlib*) demonstates how to collect user c++ functions in a separate 'library' file.

Step 1: writing user c++ functions

Justina calls user functions written in c++ (named 'user c++ functions') using the SAME Justina syntax as it uses for calling any internal Justina function, passing between 0 and 8 (eight) function arguments back and forth (values are passed by reference) and returning a function result, provided that the user c++ functions utilize the interfacing mechanism described here.

No matter the number of arguments provided by Justina when calling a user c++ function, the c++ implementation of that user function always has 4 (four) parameters.

c++ function prototype:

returnType functionName(void** const pdata, const char* const valueType, const int argCount, int& execError) ;

parameter 1: void** const pdata

pointer to an array of void pointers to (maximum eight) arguments, passed by reference by Justina.

parameter 2: const char* const valueType

pointer to an array, indicating the value types (long, float or char*) of the respective arguments, and whether these arguments are Justina variables or constants. Value types are:

`Justina::value_isLong`	32-bit signed integer
`Justina::isFloat`	32-bit floating point value
`Justina::value_isString`	character string

When checking a value type, 'bit and' it first with constant `Justina::value_typeMask` .

ValueType bit 7 indicates that the corresponding argument is a Justina variable (defined with the **var** command).

parameter 3: const int argCount

number of supplied Justina arguments, from 0 to 8.

parameter 4: int& execError

to raise a Justina error, return an error code. Justina will handle this error as it handles all other errors: Justina will stop execution unless the error is caught by the Justina **trapErrors** command.
The valid range of error codes is from **3000 to 4999**. Outside this range, error codes will be discarded.

Some of these error codes have specific meanings within Justina (see list of error codes in the user documentation) and, while all error codes in the range given are acceptable, it makes sense to attribute a meaningful error number to a specific error.

All *Justina* arguments (0 to 8) are passed by reference: Justina sets a pointer to the respective arguments (integer (c++ long), floating point (c++ float) or text (c++ char*) and passes the pointer to the user c++ function.

If an argument passed by Justina is not a variable but a constant or a Justina expression, Justina actually passes a pointer to a COPY of the value. This helps to ensure that the user c++ procedure does not inadvertently change the original value.
In case the address pointed to is an ARRAY element, the user actually has access to the complete array by setting a pointer to subsequent or preceding array elements.

Within a user c++ procedure:

- do NOT change the value type (float, character string) of an argument
- you can change the characters in a string but NEVER INCREASE the length of strings
- empty strings cannot be changed at all (this would increase the length of the string)
- it is allowed to DECREASE the length of a string (with a '\0' terminating character), but keep in mind that the string will still occupy the same amount of memory (except when you change a string to an empty string - writing a '\0' terminating character in the first position - because in Justina, empty strings do not occupy memory)
- ONLY change the (0 to 8) Justina arguments pointed to by the first c++ function argument, NOTHING ELSE.
- You can bypass checking of argument types and count if you are confident that the calling Justina function adheres to what the called c++ function expects as function arguments

Return values

User c++ functions can return a c++ Boolean, char, int, long, float, char* as a result, or nothing (void).
Justina will convert c++ Boolean, char and int return values to integers (c++: long values) upon return. c++ functions returning void: the Justina function will return zero.

Notes

- Do NOT return a char* pointing to a local c++ char array, unless you declare it as static (local variables exist on the stack until you leave the procedure, and the pointer returned to Justina may point to garbage).
- If you return an object created on the heap (NEW), make sure to save the pointer (e.g., as a static variable) because you will have to DELETE the object later (also from a user c++ procedure)
- You can return a string literal, because these strings are stored in static memory (e.g., ' *return "OK";* ')

Step 2: storing user c++ function attributes in arrays

Justina must be informed about the user c++ functions it needs to have access to. Justina needs the following:

- a function pointer (start address of the function)
- the Justina function name (name to use when calling the function from Justina). This name (alias) must follow the same Justina naming convention as for all other Justina identifiers

  Preferably, start your aliases with one of these three prefixes: **cpp_** , **usr_** or **user_** . If you use Notepad++ as Justina text editor, this will ensure proper highlighting of the Justina function name, just like any other Justina internal or user function.

- minimum (0) and maximum (8) number of arguments allowed when the user c++ function is called. The actual number of arguments supplied can then be checked when Justina parses the function call.

This information, grouped by function return type, is stored in arrays of a specific type, defined by Justina (a separate array for each function return type).

The (still empty) arrays (one for each return type):

```
Justina::CppVoidFunction   const cppVoidFunctions[]{};
Justina::CppBoolFunction   const cppBoolFunctions[]{};
Justina::CppCharFunction   const cppCharFunctions[]{};
Justina::CppIntFunction    const cppIntFunctions[]{};
Justina::CppLongFunction   const cppLongFunctions[]{};
Justina::CppFloatFunction  const cppFloatFunctions[]{};
Justina::Cpp_pCharFunction const cpp_pCharFunctions[]{};
```

In each array, create records for each user c++ function with the corresponding function return type.

A record for a user c++ function

```
{"JustinaFunctionName", functionName, minArg, maxArg}
```

Enter the c++ function name (without parentheses) to supply the function pointer.

Notes

- If there are no user c++ functions with a specific return type, you do not need to create the corresponding (empty) array.
- the Justina function name does not need to be the same name as the user c++ function name.
- Functions with invalid Justina names can not be called from Justina.

Step 3:

User c++ functions are implemented as callback functions. Justina must be informed about their existence and function attributes before Justina can call them.

You 'register' user c++ functions with a specific return type, by calling a Justina method for that return type, passing the information stored in the array for that return type.

The methods require two arguments:

- the name of the array for the respective return type
- the count of user c++ functions with this return type

Registering user c++ functions with Justina:

```
    justina.registerFloatUserCppFunctions(cppFloatFunctions, count);
    justina.register_pCharUserCppFunctions(cpp_pCharFunctions, count);
    justina.registerVoidUserCppFunctions(cppVoidFunctions, count);
    justina.registerBoolUserCppFunctions(cppBoolFunctions, count);
    justina.registerCharUserCppFunctions(cppCharFunctions, count);
    justina.registerIntUserCppFunctions(cppIntFunctions, count);
    justina.registerLongUserCppFunctions(cppLongFunctions, count);
```

Notes

- Register user c++ functions BEFORE starting the interpreter (before calling the .begin*()* method)
- If there are no user c++ functions with a specific return type, you do not need to call the corresponding Justina method.

Example programs

The Justina library contains 2 sketches containing user c++ functions.

- Justina_userCPP.ino demonstrates how to write user c++ functions for Justina
- Justina_userCPP_lib.ino demonstrates how to create a Justina user c++ 'library' file

Arduino IDE: *File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_userCPP*
File -> Examples -> Examples from custom libraries -> Justina interpreter -> Justina_userCPP_lib

Appendix F Installing Notepad++ and the Justina language extension

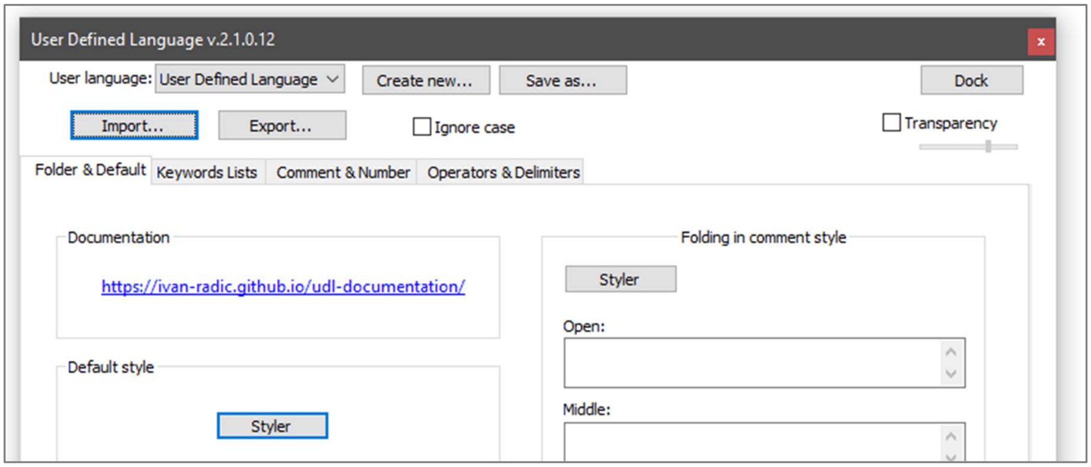
On your computer, download and install Notepad++ (https://notepad-plus-plus.org/downloads/)

Open Notepad++.

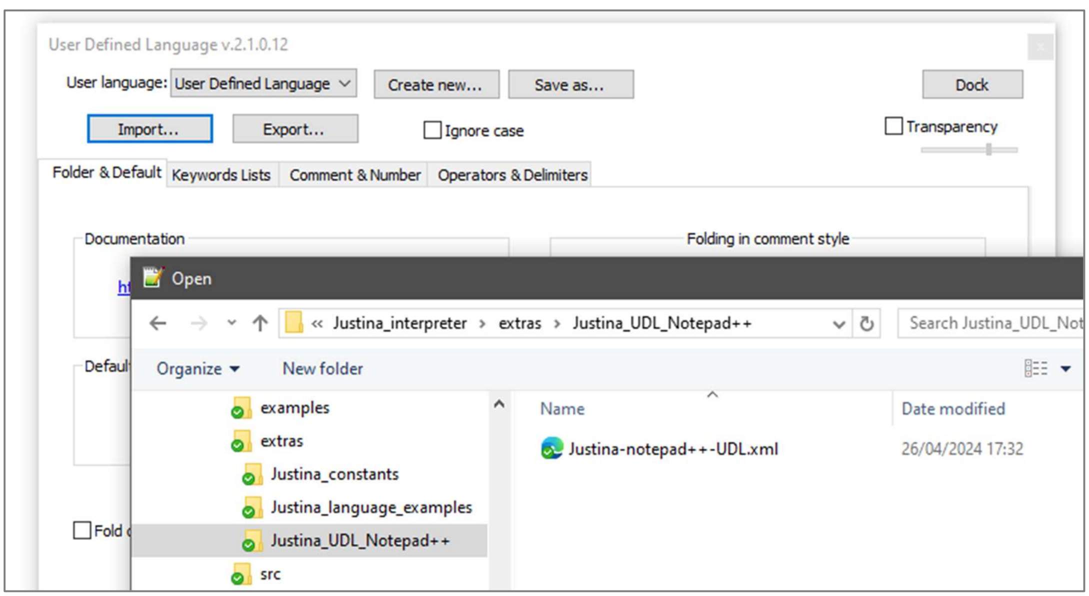
In NotePad++, select

Language -> User Defined Language -> Define your language...

A popup window will open.:



Click '*Import...*' and browse to folder *'libraries\Justina_interpreter\extras\ Justina_UDL_Notepad++'* in your Arduino sketchbook location.



Select file '*Justina_notepad++\UDL.xml*' and click *'Open'*

Close the popup

The Justina Language Extension is now installed. This means that Justina is now one of the many languages available for syntax highlighting.

Select Justina as language extension for an open file:

In Notepad++, select

Language -> Justina

Justina syntax highlighting is now enabled for the currently open file.

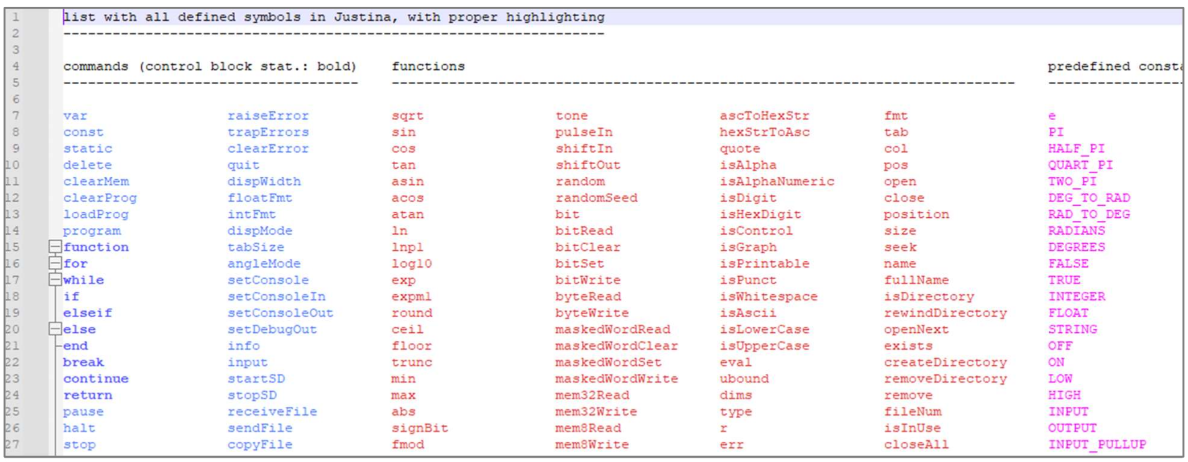
<u>Note</u>: text files ending with the '*.jus*' extension will automatically select the Justina Language Extension on opening.

⇒ **Use .jus as extension for your Justina programs**

Checking that the Justina extension is properly installed

In Notepad++, open file 'test_highlight.jus' in folder *'libraries\Justina_interpreter\extras\ Justina_UDL_Notepad++'* (in your Arduino sketchbook location).

The opened file does not contain a program but merely the collection of all words and symbols (command names, function names, operators, predefined constants) recognized by Justina, with proper highlighting.



```
list with all defined symbols in Justina, with proper highlighting
------------------------------------------------------------------

commands (control block stat.: bold)    functions                                                                    predefined const
------------------------------------    ----------------------------------------------------------------------------    ----------------

var          raiseError      sqrt     tone             ascToHexStr      fmt               e
const        trapErrors      sin      pulseIn          hexStrToAsc      tab               PI
static       clearError      cos      shiftIn          quote            col               HALF_PI
delete       quit            tan      shiftOut         isAlpha          pos               QUART_PI
clearMem     dispWidth       asin     random           isAlphaNumeric   open              TWO_PI
clearProg    floatFmt        acos     randomSeed       isDigit          close             DEG_TO_RAD
loadProg     intFmt          atan     bit              isHexDigit       position          RAD_TO_DEG
program      dispMode        ln       bitRead          isControl        size              RADIANS
function     tabSize         lnp1     bitClear         isGraph          seek              DEGREES
for          angleMode       log10    bitSet           isPrintable      name              FALSE
while        setConsole      exp      bitWrite         isPunct          fullName          TRUE
if           setConsoleIn    expm1    byteRead         isWhitespace     isDirectory       INTEGER
elseif       setConsoleOut   round    byteWrite        isAscii          rewindDirectory   FLOAT
else         setDebugOut     ceil     maskedWordRead   isLowerCase      openNext          STRING
end          info            floor    maskedWordClear  isUpperCase      exists            OFF
break        input           trunc    maskedWordSet    eval             createDirectory   ON
continue     startSD         min      maskedWordWrite  ubound           removeDirectory   LOW
return       stopSD          max      mem32Read        dims             remove            HIGH
pause        receiveFile     abs      mem32Write       type             fileNum           INPUT
halt         sendFile        signBit  mem8Read         r                isInUse           OUTPUT
stop         copyFile        fmod     mem8Write        err              closeAll          INPUT_PULLUP
```

Some of the Justina commands (blue and dark blue), functions (red) and predefined constants (magenta) as shown in Notepad++ with the Justina language extension installed

Appendix G Installing YAT terminal

The Arduino IDE Serial Monitor, although a great tool for uploading your compiled Arduino programs and for communicating with your Arduino (and Justina), does not allow sending ***files*** to Justina. As a Justina program consists of a text file that is edited on your computer (preferably with Notepad++), there are only two ways to load and parse a Justina program in your Arduino:

1. If an SD card module is hooked up to your Arduino, you can copy program files from your computer to an SD card, and then insert that SD card in the Arduino SD card board. But this means that you constantly need to insert and remove SD cards. And there's always a risk that during one of these operations your SD card will get corrupted.
2. Send the program file to your Arduino via Serial, a TCP client, ... and either store it on an SD card to load it from there, or load and parse the program immediately while it's being sent.

While the second option is the most straightforward one (especially if you go through a series of program load, test, debug, correct and reload ... iterations), the Arduino IDE Serial Monitor doesn't support that.

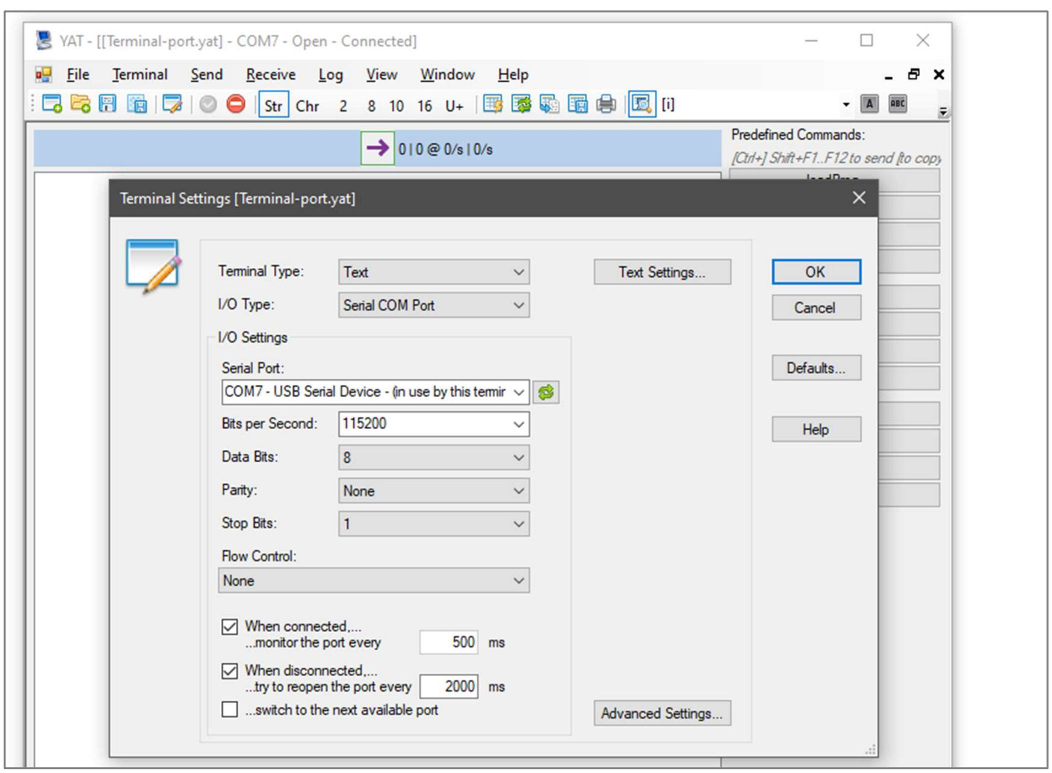
Fortunately, there are several good free terminal programs available. The one I prefer is YAT and we use it throughout most examples in this manual. A second one which works quite well is named Tera Term. These terminal programs can be freely downloaded on your PC. They allow for serial communication via USB as well as via TCP / IP connections.

In what follows, we'll stick to YAT because it has a couple of nice, useful features.

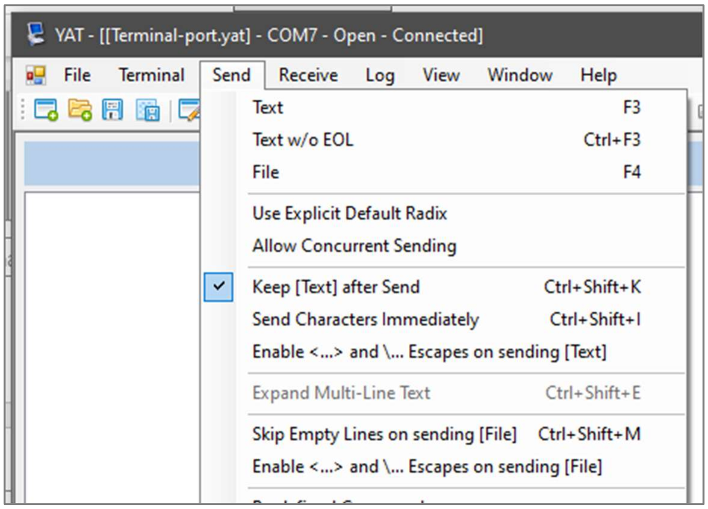
Download and install YAT

On your computer, download and install YAT (https://sourceforge.net/projects/y-a-terminal/).

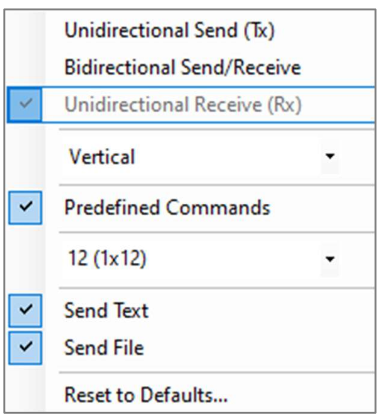
Under 'Terminal->settings', select the USB port the Arduino is connected to, the baud rate etc. and press OK.



In the YAT menu, select '*Send*' and, in the dropdown that will open, verify that the only option selected is '*Keep [Text] after Send*'. Select it if needed and deselect all the other options (if selected).



Now select '*View->panels*' and, in the dropdown that will open, verify that the options selected are as indicated in the figure. Deselect the other options (if selected).



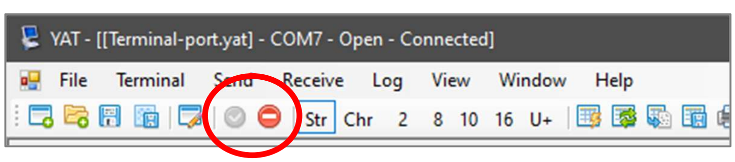
Yat will now only display characters it receives from your Arduino and will not echo any characters it sends to Arduino (Justina will take care of echoing characters it receives from YAT).

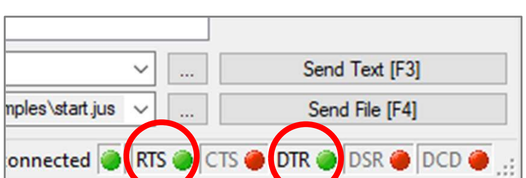
Sending text and files to Arduino is now enabled as well, as is the use of predefined commands.

In the '*View*' menu as well, you might want to disable formatting (it's only overloading what you see).

Connecting / disconnecting YAT

Connect and disconnect YAT, using the two buttons indicated.



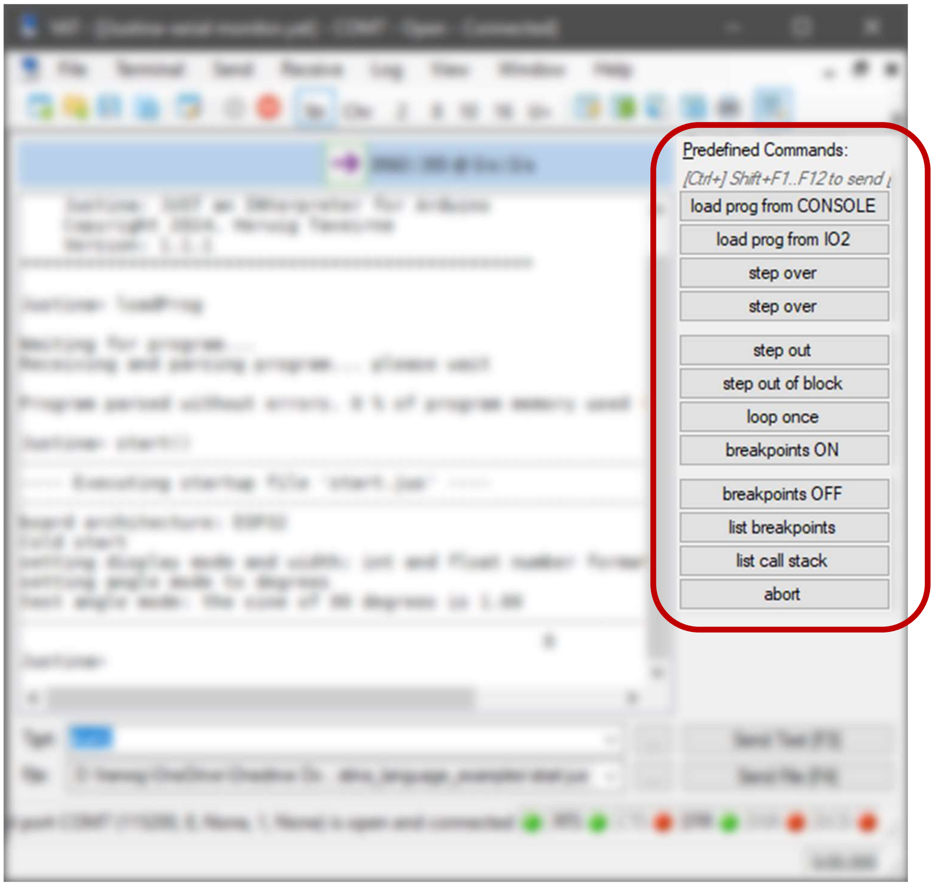


While connected, verify that indicators 'RTS' (Request to Send) and 'DTR' (Data Terminal Ready) are ON (green light). Click on the indicators to change their status, if currently OFF (red lights).
The other indicators are not relevant here.

Predefined commands

One final, great feature of YAT is that you can enter a set of Predefined Commands, accessible via a number of buttons and saved together with the other terminal settings (button *'Save Terminal'*, underneath the YAT menu).

When clicking a button, the corresponding predefined command will be sent to Justina.



Example of a predefined Justina command set

You can now use YAT as your serial monitor to send Justina statements to your Arduino (type a statement in the 'Send Text' textbox and press Enter or F3) and see your Arduino's response, as you did in the preceding examples.

⚠	Remember to close the Arduino Serial Monitor before connecting the Terminal app to your Arduino., and vice versa

Sending binary files

To send a binary file with YAT, you'll have to temporarily change the Terminal type from Text to Binary (in terminal settings).

If you need to send commands as well (e.g., '**receiveFile** "image001.jpg" '), you must enable 'Escape sequences on sending text' (in the Send menu) and terminate your commands with ' \n' (in binary mode, YAT will not add a newline character by itself).

Appendix H List of predefined constants

Symbolic constant name	value	Data type	Description
Math symbols			
e	2.71828182...	floating point	base of natural logarithm
PI	3.14159265...	floating point	Pi
HALF_PI	1.57079632...	floating point	π / 2
QUART_PI	0.78539816...	floating point	π / 4
TWO_PI	6.28318530...	floating point	2 π
Conversion factors			
DEG_TO_RAD	0.01745329...	floating point	degrees to radians
RAD_TO_DEG	57.2957795...	floating point	radians to degrees
AngleMode command: setting angle mode			
RADIANS	0	integer	Angle mode set to radians
DEGREES	1	integer	Angle mode set to degrees
Boolean constants			
FALSE	0	integer	
TRUE	1	integer	
OFF	0	integer	
ON	1	integer	
LOW	0	integer	
HIGH	1	integer	
Arduino Digital IO			
INPUT	0x1	integer	pinMode
OUTPUT	0x3	integer	pinMode
INPUT_PULLUP	0x5	integer	pinMode
INPUT_PULLDOWN	0x9	integer	pinMode
LSBFIRST	0	integer	shiftOut, shiftIn: bitOrder
MSBFIRST	1	integer	shiftOut, shiftIn: bitOrder
LED_BUILTIN	13	integer	built-in LED pin
LED_RED	14	integer	nano ESP32 only: red LED pin
LED_GREEN	15	integer	nano ESP32 only: green LED pin
LED_BLUE	16	integer	nano ESP32 only: blue LED pin
Checking data types: type function result			
INTEGER	1	integer	argument type is integer
FLOAT	2	integer	argument type is floating point
STRING	3	integer	argument type is string
Console display mode: dispMode command arguments			
NO_PROMPT	0	integer	no prompt, no echo
PROMPT	1	integer	show prompt but no echo
ECHO	2	integer	show prompt and echo input
NO_RESULTS	0	integer	do not display results
RESULTS	1	integer	display results
QUOTE_RES	2	integer	display results; strings between quotes
info statement: 'flag' argument (entry)			
ENTER	0	integer	'Enter' answer accepted
ENTER_CANCEL	1	integer	'Enter' and 'Cancel' answer accepted
YES_NO	2	integer	'Yes' and 'No' answer accepted
YN_CANCEL	3	integer	'Yes', 'No' and 'Cancel' answer accepted
input statement: 'flag' argument (entry)			
NO_DEFAULT	0	integer	'Default' is not allowed as answer
ALLOW_DEFAULT	1	integer	'Default' is allowed as answer

Symbolic constant name and value		Data type	Description
Additional test values			
CANCELED	0	integer	Operation was canceled
OK	1	integer	Operation confirmed
NOK	-1	integer	Operation was not confirmed
External device IO			
CONSOLE	0	integer	input & output from / to console
IO1	-1	integer	input & output from / to IO device 1
IO2	-2	integer	input & output from / to IO device 2
IO3	-3	integer	input & output from / to IO device 3
IO4	-4	integer	input & output from / to IO device 4
File IO			
FILE1	1	integer	input & output from / to file number 1
FILE2	2	integer	input & output from / to file number 2
FILE3	3	integer	input & output from / to file number 3
FILE4	4	integer	input & output from / to file number 4
FILE5	5	integer	input & output from / to file number 5
READ	0x1	integer	file mode
WRITE	0x2	integer	file mode
APPEND	0x6	integer	file mode
SYNC	0x8	integer	file mode
NEW_OK	0x10	integer	file mode
NEW_ONLY	0x30	integer	file mode
TRUNC	0x40	integer	file mode
EOF	-1	integer	use to indicate 'EOF' position
Formatting: floating point notation			
FIXED	"f"	string	fixed point notation
EXP	"e"	string	scientific notation
EXP_U	"E"	string	scientific notation, 'E' uppercase
SHORT	"g"	string	shortest notation (fixed or scientific)
SHORT_U	"G"	string	shortest notation: 'E' uppercase
Formatting: integer number notation			
DEC	"d"	string	decimal representation
HEX	"x"	string	hexadecimal representation
HEX_U	"X"	string	hexadecimal, A..F uppercase
Formatting: character strings			
CHARS	"s"	string	character string
Formatting: flags			
FMT_LEFT	0x01	integer	align output left within the print field
FMT_SIGN	0x02	integer	numeric values: always add sign
FMT_SPACE	0x04	integer	numeric values: add space if no sign
FMT_POINT	0x08	integer	floating point only: always add decimal point
FMT_0X	0x08	integer	hex notation only: add 0x or 0X if not zero
FMT_000	0x10	integer	floating point only: pad print field with zeros
FMT_NONE	0x00	integer	clear all flags
Arduino board information			
BOARD_OTHER	0	integer	non-supported board
BOARD_SAMD	1	integer	Arduino SAMD architecture
BOARD_RP2040	2	integer	Arduino nano RP2040
BOARD_ESP32	3	integer	Arduino nano ESP32 only

Appendix I Error codes

Error number	Error code
0	noerror
	Parsing errors
1000	result_statementTooLong
1001	result_tokenNotFound
1002	result_missingLeftParenthesis
1003	result_missingRightParenthesis
1100	result_separatorNotAllowedHere
1101	result_operatorNotAllowedHere
1102	result_prefixOperatorNotAllowedhere
1103	result_invalidOperator
1104	result_parenthesisNotAllowedHere
1105	result_resWordNotAllowedHere
1106	result_functionNotAllowedHere
1107	result_variableNotAllowedHere
1108	result_alphaConstNotAllowedHere
1109	result_numConstNotAllowedHere
1110	result_assignmNotAllowedHere
1111	result_cannotChangeConstantValue
1112	result_identifierNotAllowedHere
1113	result_prefixCharNotAllowedHere
1200	result_constantValueExpected
1201	result_variableNameExpected
1202	result_assignmentOrSeparatorExpected
1203	result_separatorExpected
1300	result_maxVariableNamesReached
1301	result_maxLocalVariablesReached
1302	result_maxStaticVariablesReached
1303	result_maxJustinaFunctionsReached
1304	result_progMemoryFull
1400	result_identifierTooLong
1401	result_spaceMissing
1402	result_token_not_recognised
1403	result_alphaConstTooLong
1404	result_alphaConstInvalidEscSeq
1405	result_alphaNoCtrlCharAllowed
1406	result_alphaClosingQuoteMissing
1407	result_numberInvalidFormat
1408	result_parse_overflow
1500	result_function_wrongArgCount
1501	result_function_redefinitionNotAllowed
1502	result_function_mandatoryArgFoundAfterOptionalArgs
1503	result_function_maxArgsExceeded
1504	result_function_prevCallsWrongArgCount
1505	result_function_defsCannotBeNested
1506	result_function_scalarAndArrayArgOrderNotConsistent
1507	result_function_scalarArgExpected
1508	result_function_arrayArgExpected
1509	result_function_redefiningNotAllowed
1510	result_function_undefinedFunctionOrArray

Error number	Error code
1511	result_function_arrayParamMustHaveEmptyDims
1512	result_function_needsParentheses
1600	result_var_nameInUseForFunction
1601	result_var_notDeclared
1602	result_var_redeclared
1603	result_var_definedAsScalar
1604	result_var_definedAsArray
1605	result_var_constantArrayNotAllowed
1606	result_var_constantVarNeedsAssignment
1607	result_var_ControlVarInUse
1608	result_var_controlVarIsConstant
1609	result_var_illegalInDeclaration
1610	result_var_illegalInProgram
1611	result_var_usedInProgram
1612	result_var_deleteSyntaxinvalid
1700	result_arrayDef_noDims
1701	result_arrayDef_negativeDim
1702	result_arrayDef_dimTooLarge
1703	result_arrayDef_maxDimsExceeded
1704	result_arrayDef_maxElementsExceeded
1705	result_arrayDef_emptyInitStringExpected
1706	result_arrayDef_dimNotValid
1707	result_arrayUse_noDims
1708	result_arrayUse_wrongDimCount
1800	result_cmd_programCmdMissing
1801	result_cmd_onlyProgramStart
1802	result_cmd_onlyImmediateMode
1803	result_cmd_onlyImmModeFirstStatement
1804	result_cmd_onlyInsideProgram
1805	result_cmd_onlyInsideFunction
1806	result_cmd_onlyOutsideFunction
1807	result_cmd_onlyImmediateOrInFunction
1808	result_cmd_onlyInProgOutsideFunction
1809	result_cmd_onlyImmediateNotWithinBlock
1810	result_cmd_resWordExpectedAsPar
1811	result_cmd_expressionExpectedAsPar
1812	result_cmd_varWithoutAssignmentExpectedAsPar
1813	result_cmd_varWithOptionalAssignmentExpectedAsPar
1814	result_cmd_variableExpectedAsPar
1815	result_cmd_variableNameExpectedAsPar
1816	result_cmd_identExpectedAsPar
1817	result_cmd_argumentMissing
1818	result_cmd_tooManyArguments
1900	result_userCB_allAliasesSet
1901	result_userCB_aliasRedeclared
2000	result_block_noBlockEnd
2001	result_block_noOpenBlock
2002	result_block_noOpenLoop
2003	result_block_noOpenFunction
2004	result_block_notAllowedInThisOpenBlock
2005	result_block_wrongBlockSequence
2100	result_trace_eval_resWordNotAllowed

Error number	Error code
2101	result_trace_eval_genericNameNotAllowed
2102	result_trace_userFunctonNotAllowed
2103	result_trace_evalFunctonNotAllowed
2104	result_parseList_stringNotComplete
2105	result_parseList_valueToParseExpected
2106	result_BP_lineRangeTooLong
2107	result_BP_lineTableMemoryFull
2108	result_BP_emptyTriggerString
2109	result_BP_triggerString_nothingToEvaluate
2200	result_parse_abort
2201	result_parse_setStdConsole
2202	result_parse_kill
	Execution errors
3000	result_array_subscriptOutsideBounds
3001	result_array_subscriptNonInteger
3002	result_array_subscriptNonNumeric
3003	result_array_dimCountInvalid
3004	result_array_valueTypeIsFixed
3100	result_arg_outsideRange
3101	result_arg_integerTypeExpected
3102	result_arg_floatTypeExpected
3103	result_arg_stringExpected
3104	result_arg_numberExpected
3105	result_arg_nonEmptyStringExpected
3106	result_arg_stringTooShort
3107	result_arg_invalid
3108	result_arg_integerDimExpected
3109	result_arg_dimNumberInvalid
3110	result_arg_variableExpected
3111	result_arg_tooManyArgs
3112	result_arg_wrongSpecifierForDataType
3200	result_integerTypeExpected
3201	result_floatTypeExpected
3202	result_numberExpected
3203	result_operandsNumOrStringExpected
3204	result_undefined
3205	result_overflow
3206	result_underflow
3207	result_divByZero
3208	result_testexpr_numberExpected
3300	result_noProgramStopped
3400	result_BP_sourcelineNumberExpected
3401	result_BP_notAllowedForSourceLine
3402	result_BP_statementIsNonExecutable
3403	result_BP_maxBPentriesReached
3404	result_BP_wasNotSet
3405	result_BP_hitcountNotWithinRange
3406	result_BP_sourceLineNotInStoppedFunction
3407	result_BP_cannotMoveIntoBlocks
3500	result_EVAL_emptyString=3500
3501	result_EVAL_nothingToEvaluate

Error number	Error code
3502	result_EVAL_parsingError
3503	result_list_parsingError
3600	result_SD_noCardOrNotAllowed
3601	result_SD_noCardOrCardError
3602	result_SD_fileNotFound
3603	result_SD_couldNotOpenFile
3604	result_SD_fileIsNotOpen
3605	result_SD_fileAlreadyOpen
3606	result_SD_invalidFileNumber
3607	result_SD_fileIsEmpty
3608	result_SD_maxOpenFilesReached
3609	result_SD_fileSeekError
3610	result_SD_directoryExpected
3611	result_SD_directoryNotAllowed
3612	result_SD_couldNotCreateFileDir
3613	result_SD_directoryDoesNotExist
3614	result_SD_pathIsNotValid
3615	result_SD_sourceIsDestination
3616	result_SD_fileNotAllowedHere
3700	result_IO_invalidStreamNumber
3701	result_IO_noDeviceOrNotForInput
3702	result_IO_noDeviceOrNotForOutput

Appendix J Justina Command and Function index

This index lists all Justina commands and built-in functions, along with the page numbers where they appear. Commands are shown in **bold**, functions in *italic*.