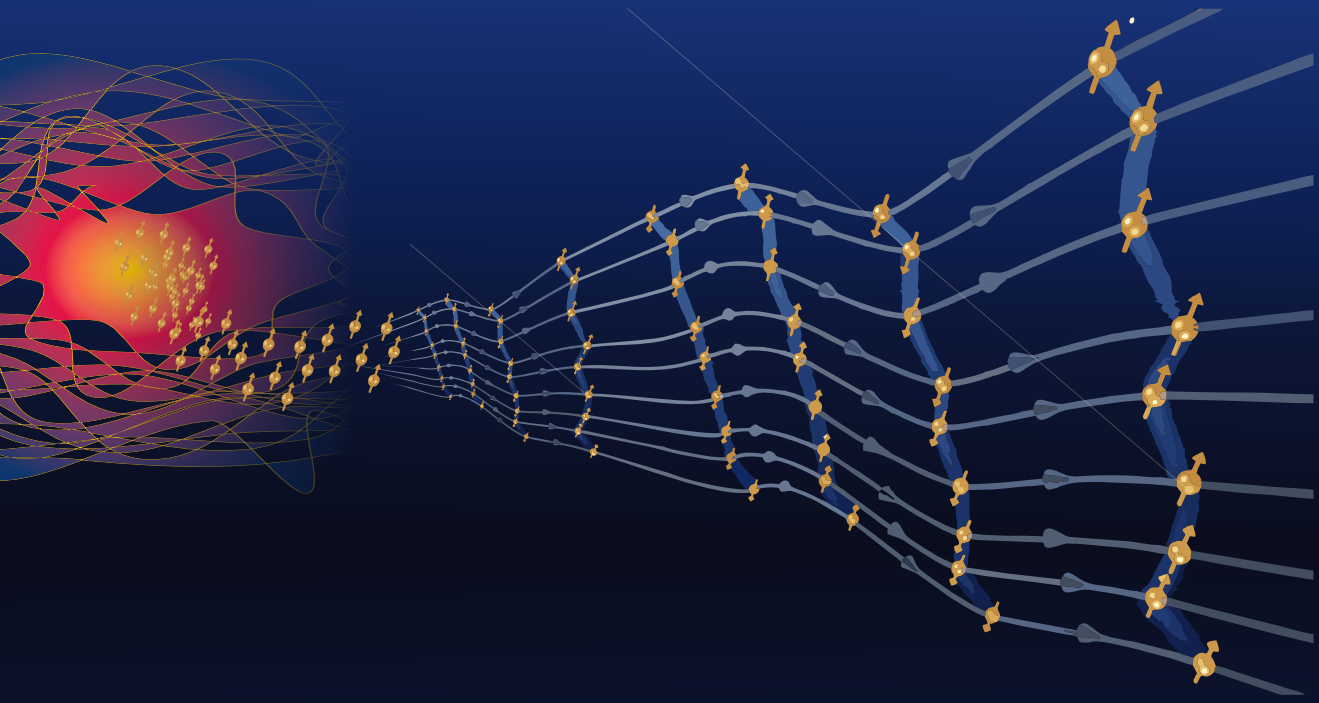
BLAISE PASCAL MAGAZINE 101

Multi platform /Object Pascal / Internet / JavaScript / WebAssembly / Pas2Js / Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux



Faker: Synthetic Data Generator
Migration Guide to Firebird 4.0
PAS2JS Communicating with the webserver (Part 2)
Polygons in the making
Raspberry Pi with Windows 11 / Delphi & Lazarus running
Webassembly for PAS2JS

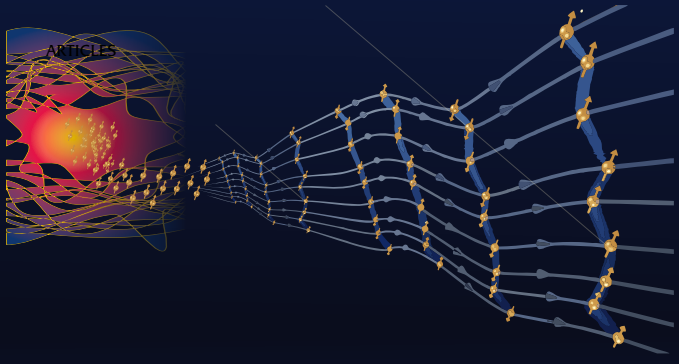
BLAISE PASCAL MAGAZINE 101

Multi platform /Object Pascal / Internet / JavaScript / WebAssembly / Pas2Js / Databases / CSS Styles / Progressive Web Apps Android / IOS / Mac / Windows & Linux

Blaise Pascal

CONTENT

ARTICLES



Time cristals

In condensed matter physics, a time crystal is a quantum system of particles whose lowest-energy state is one in which the particles are in repetitive motion. The system cannot lose energy to the environment and come to rest because it is already in its quantum ground state.

Because of this the motion of the particles does not really represent kinetic energy like other motion, it has "motion without energy".

Time crystals were first proposed theoretically by Frank Wilczek in 2012 as a time-based analogue to common crystals — whereas the atoms in crystals are arranged periodically in space, the atoms in a time crystal are arranged periodically in both space and time.

Several different groups have demonstrated matter with stable periodic evolution in systems that are periodically driven. In terms of practical use, time crystals may one day be used as quantum memories.

ADVERTISERS

Niklaus Wirth

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed (left below) in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. A derivative known as Object Pascal designed for object-oriented programming was developed in 1985. The language name was chosen to honour the Mathematician, Inventor of the first calculator: Blaise Pascal (see top right).

Publisher: PRO PASCAL FOUNDATION in collaboration © Stichting Ondersteuning Programmeertaal Pascal - Netherlands

Stephen Ball
http://delphiaball.co.uk
@DelphiABall

Dmitry Boyarintsev
dmitry.living @ gmail.com

Michaël Van Canneyt,
michael @ freepascal.org

Marco Cantù
www.marcocantu.com
marco.cantu @ gmail.com

David Dirkse
www.davdata.nl
E-mail: David @ davdata.nl

Benno Evers
b.evers @ everscustomtechnology.nl

Bruno Fierens
www.tmssoftware.com
bruno.fierens @ tmssoftware.com

Holger Flick
holger @ flixments.com

Mattias Gärtner
nc-gaertnma@netcologne.de

Max Kleiner
www.softwareschule.ch
max @ kleiner.com

John Kuiper
john_kuiper @ kpnmail.nl

Wagner R. Landgraf
wagner @ tmssoftware.com

Vsevolod Leonov
vsevolod.leonov@mail.ru

Andrea Magni www.andreamagni.eu
andrea.magni @ gmail.com
www.andreamagni.eu/wp

Paul Nauta PLM Solution Architect
CyberNautics
paul.nauta @ cybernautics.nl

Kim Madsen
www.component4developers.com

Boian Mitov
mitov @ mitov.com

Jeremy North
jeremy.north @ gmail.com

Detlef Overbeek - Editor in Chief
www.blaisepascal.eu
editor @ blaisepascal.eu

Howard Page Clark
hdpc @ talktalk.net

Heiko Rompel
info @ rompelsoft.de

Wim Van Ingen Schenau -Editor
wisone @ xs4all.nl

Rik Smit
rik @ blaisepascal.eu

Bob Swart
www.eBob42.com
Bob @ eBob42.com

B.J. Rao
contact @ intricad.com

Daniele Teti
www.danieleteti.it
d.teti @ bittime.it

Anton Vogelaar
ajv @ vogelaar-electronics.com

Danny Wind
dwind @ delphicompany.nl

Jos Wegman / Corrector / Analyst

Siegfried Zuhr
siegfried @ zuhr.nl

Editor - in - chief
Detlef D. Overbeek, Netherlands Tel.: Mobile: +31 (0)6 21.23.62.68
News and Press Releases email only to **editor@blaisepascal.eu**

Editors
Peter Bijlsma, W. (Wim) van Ingen Schenau, Rik Smit

Correctors
Howard Page-Clark, Peter Bijlsma

Subscriptions (2019 prices)

	Internat. excl. VAT	Internat. incl. 9% VAT	Shipment
Printed Issue ±60 pages	€ 155,96	€ 250	€ 80,00
Electronic Download Issue 60 pages	€ 64,20	€ 70	———
Printed Issue inside Holland (Netherlands) 60 pages	———	€ 250,00	€ 70,00

Subscriptions can be taken out online at **www.blaisepascal.eu** or by written order, or by sending an email to **office@blaisepascal.eu**
Subscriptions can start at any date. All issues published in the calendar year of the subscription will be sent as well.
Subscriptions run 365 days. Subscriptions will not be prolonged without notice. Receipt of payment will be sent by email.
Subscriptions can be paid by sending the payment to:
ABN AMRO Bank Account no. 44 19 60 863 or by credit card or Paypal
Name: Pro Pascal Foundation-Foundation for Supporting the Pascal Programming Language (Stichting Ondersteuning Programeertaal Pascal)
IBAN: NL82 ABNA 0441960863 BIC ABNANL2A VAT no.: 81 42 54 147 (Stichting Programmeertaal Pascal)
Subscription department
Edelstenenbaan 21 / 3402 XA IJsselstein, The Netherlands
Mobile: + 31 (0) 6 21.23.62.68 **office@blaisepascal.eu**

Member and donator of WIKIPEDIA
Member of the **Royal Dutch Library**
KB

From your editor

A happy new year to everybody!
Finally there seems to be some light at the end of this tunnel.
We were able to finalize some very important wishes:
Lazarus had a new update and for Free Pascal we have been able to add some very special items: Generics (already integrated in the FPC-TrunkVersion), and Anonymous functions should become available very soon now.

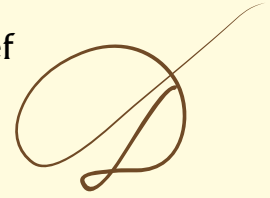
We added WebAssembly as you can read in this issue (*Webassembly for PAS2JS page 85*) and for PAS2JS we have started with a series of article (*lessons*) see page: *"PAS2JS Communicating with the webserver (Part 2) - starting at page 20* that will later become a book.
We created a new Mini Server for Testing Purposes which will be shown in the next item. That will make it very easy to built web-sites in PAS2JS and also create desktop applications which will run in your browser and show you how to do so.

I had in mind to do much more items in this issue but because the articles already added up to 100 pages,
I decided to publish them in the next issue 102.
Since wee now have WebAssembly we will create a web-store which will be capable of creating shop-connections with banks and other module providers (*like we use Molly*) in Pascal and show this sample code so you all could use that.
This is the basis for our new to build website.
I think is ridiculous that we run a site that is not build on Pascal.
It also has a very nice learning aspect which will demonstrate very well what potential PAS2JS and WebAssembly has.

I am already planning the next real-life meetings. I'll tell you soon...
Might be beginning April 2022..

I hope this year will become a very interesting and successful year as ever for you...

Detlef



"Our smart refrigerator just texted a 'Glacier Alert.'"

LibraryForm

Show article | Go to issue Nr | Issue | Search

- Advertisement -

ID	IssueNr	Author	Article	PDF	PageNr
861	99_100	Michael van Canneyt	GIT continued: Branching and partial commits for Lazarus and Delphi		36
862	99_100	Bob Swart	Testing IntraWeb Applications with Delphi		145
863	99_100	Paulo César Botelho Barbosa	Skia for Delphi		30
864	99_100	Detlef Overbeek	Creating a new Library Program with PDF viewer		157
865	99_100	Michael van Canneyt	Introduction to programming the internet with PAS2S		98
866	99_100	Bruno Fierens	Delphi for Raspberry		55
867	99_100	Danny Wind	Last part of the Webservice (5), Deploy to Apache		124
868	99_100	Den Zubow	PDF document in a report – using of new TfrxPDFView object		114
869	99_100	Newscientist and Detlef Overbeek	Scaling to a Very Large AI (Artificial Intelligence) causes unprecedented insights		169
870		Detlef	test		

Show Thumbnails | Page | Jump to page | server

The new Library stick program has arrived with some improvements..

❶ The thumbnails are created in the background so they load much faster.

❷ In the Image you can see a large text field where you can search in all the text of that issue.
The pages that are relevant will appear at th bottom. By clicking on the item you will be guided to the page.

Figure 1: Designing HTML in a RAD manner JSON

Page 23: ["...\" your changes to a server. In our case, the s..."]
Page 31: ["...hics. Highlight the server side, that is, be i...","...ow a multithreading server will be able to rec..."]
Page 77: ["...he Microsoft Speech Server Platform. There are...","... client, server, and mobile version...","... operating systems; server voices are availabl..."."...wnload for use with server applications such a..."
Page 99: ["...e 99/100 2021 99v A SERVER-SIDE BACKENDMost We..."."...dized JSON format.A server that exchanges data..."]
Page 100: ["...s a complete client server ORM/SOA framework!,...","... application server. This can be a Fast...","...CGI server, a CGI, ...","...stand-alone HTTP(s) server. ...","... a RPC
Page 101: ["...t you do not need a server to generate the pag..."]

The new Library stick program has arrived with some improvements..

1. The thumbnails are created in the background so they load much faster.
2. In the Image you can see a large text field where you can search in all the text of that issue.
   The pages that are relevant will appear at th bottom. By clicking on the item you will be guided to the page.

Blaise Library Program
+
USB Librarystick

Containing:
installer for Windows
Issues 1-100 / 5809 Pages
873 Articles / Code samples

RASPBERRY PI APPS WITH DELPHI

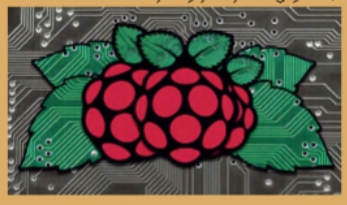
A new approach to create Raspberry Pi apps with Delphi via MILETUS

By Bruno Fierens

MILETUS

In **TMS WEB Core v1.7**, we introduced the **Miletus** technology for the first time as a technology to create cross-platform desktop applications from the **Delphi** IDE based on web technology.
So far, with **Miletus,** it was possible to create **Windows** (Win32 & Win64), macOS (Intel & ARM) and **LINUX 64** bit applications from the same codebase.
Now, from **TMS WEB CORE V1.9**, it is possible to generate applications from the **Delphi** IDE that can run on a **Raspberry Pi** using the official Raspberry Pi OS.

https://arstechnica.com/information-technology/2013/03/how-two-volunteers-built-the-raspberry-pis-operating-system/



GETTING STARTED

To create a **Raspberry Pi** app from the **Delphi IDE**, follow File | New | Other and under TMS WEB, you find the application type **TMS Miletus app.**
After the IDE created the default application, you can start adding your code to the application in pretty much the same way as you would do for a **VCL application or FireMonkey** application.
The components you can use are the same components as for a regular **TMS WEB Core** web client application, i.e. the TWEB* components.

FAKER Python4Delphi

AUTHOR: MAX KLEINER Try finally begin. – Max

Make the fake.

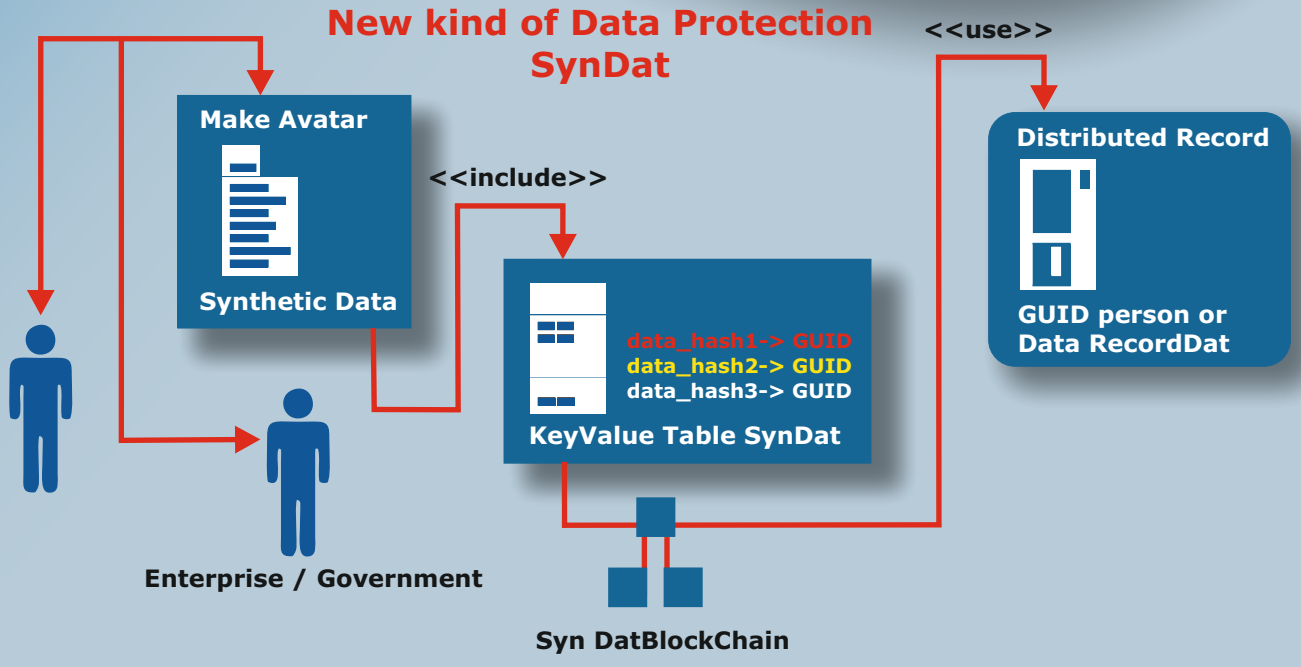
INTRODUCTION

Real data, extracted from the real world, is a gold standard for data science and data protection, perhaps for obvious reasons. In such a case, synthetic data producing can be used either in place of real data, protect real user as an avatar or to augment an insufficiently large dataset. With **Python4Delphi** scripting.

http://www.softwareschule.ch/examples/pydemo32_2.txt

Faker is a **Python** library that generates fake data.
Fake data is often used for testing or filling databases with some dummy data.
Faker is strong inspired by PHP's Faker, Perl's Data::Faker, and Ruby's Faker.

We are also able to sample from a model and create synthetic data, hence the name **SynDat.** The most obvious way that the use of synthetic data benefits data science is that it reduces the need to capture data from real-world events, and for this reason it becomes possible to generate data and construct a **dataset** much more quickly than a **dataset** dependent on real-world events and in addition you don't misuse data protection.



Now I want to show almost step by step how we can use the Faker Lib. First you had to install faker package, it can be installed with pip:

```
C:\Users\Max\AppData\Local\Programs\Python\Python36-32>
                                python -m pip install faker
```

Install a 32 bit package module in a 64 bit environment:

❶ Change to your 32 bit path with cd:
`C:\Users\Max\AppData\Local\Programs\Python\Python36-32>`
❷ Call the Pip (*e.g. faker module*) explicitly with `python.exe`: python -m pip install faker

And it runs:
Downloading `https://files.pythonhosted.org/packages/27/ab/0371598513e8179d9053911e814c4de4ec2d0dd47e725dca40aa664f994c/Faker-9.9.0-py3-none-any.whl (1.2MB)..`

You are using **pip version 9.0.1**, however version **21.3.1** is available.
You should consider upgrading via the `'python -m pip install --upgrade pip'`.
`C:\Users\Max\AppData\Local\Programs\Python\Python36-32>`

Now we start the program:
The `fake.Faker` (fake = Faker()) creates and initializes a faker generator, which can generate data by accessing properties named after the type of data, whether you need to bootstrap your database, create structured **JSON** documents or fill-in your storage persistence to stress test.

```
sw:= TStopWatch.Create();
sw.Start;
eg.execStr('from faker import Faker');
eg.execStr('import simplejson as json');  //# instead import json
eg.execStr('import dumper');
eg.execStr('fake = Faker()');
fprofile:= eg.evalStr('(fake.profile())')
fprofile:= StringReplace(fprofile,'\n',CRLF,[rfReplaceAll]);
```

To clean up the data, we will also replace the newlines as \n in the generated addresses with commas or **CRLF** (linefeeds), and remove the newlines from profile generated text completely.

Faker delegates the data generation to providers.
The default provider uses the `English locale`. **Faker** supports other locales;
they differ in level of completion, there are lots of ways to artificially manufacture and build data, some of which are far more complex than others and models real-world distribution with descriptive statistics.
Check the output with path and list the profile dictionary, the example outputs a fake name, address, and many more items of a persons profile: (*Next page*)

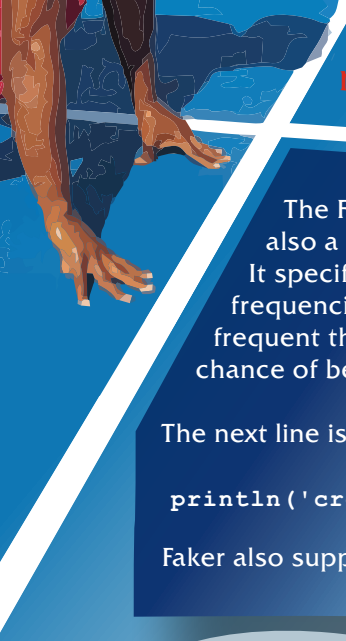
```
 fake person profile:
 {'job': 'Manufacturing engineer', 'company':
 'Cunningham-Young',
 'ssn': '630-62-0344',
 'residence': 'PSC 1590, Box 0125 APO AA 42693',
 'current_location': (Decimal('-51.8228245'), Decimal('-61.889364')),
 'blood_group': 'A+', 'website': ['http://www.jones-clark.net/',
 'https://www.fowler.com/'], 'username': 'garciatina',
 'name': 'Roger Nichols', 'sex': 'M',
 'address': '51574 Combs Alley Apt. 142, Ryanhaven, AL 82796',
 'mail': 'andrea31@hotmail.com', 'birthdate': datetime.date(1914, 4, 15)}
creditcard#: 213140049750943
Stop Watch Faker Tester1: 0:0:0.636
```

This is not json as I first assumed, and we can convert it. I tried first with json and simplejson, got some date and decimals serialize exceptions (Object of type date is not JSON serializable.), then I used dumper lib, but got a next exception Exception: <class 'AttributeError'>: 'NoneType' object has no attribute 'write'.: So the profile is a dict type, the misleading {} trapped me first.
Let's generate another avatar:

```
{'job': 'Nurse, adult',
'company': 'Rogers and Sons',
'ssn': '038-06-4652',
'residence': 'PSC 8856, Box 2882 APO AE 08426',
'current_location': (Decimal('16.4363075'), Decimal('-83.079826')),
'blood_group': 'A-',
'website': ['https://www.white.biz/', 'http://garrett-perez.com/'],
'username': 'xnelson',
'name': 'Ms. Colleen Bowman PhD',
'sex': 'F',
'address': '328 Reeves Estates Apt. 279 Lake Nicholas, MD 31753',
'mail': 'kkhan@yahoo.com',
'birthdate': datetime.date(1936, 6, 3)}
```

Oh what as surprise a nurse and she holds a PhD and works by Rogers.
What if, for instance, I'm interested in generating German or Spanish names and professions of the type one would find in Netherlands, Mexico, Austria or Switzerland?

```
fake = Faker(['de_DE'])
for i in range(10):
     print(fake.name())
eg.execStr('fake = Faker(["es_MX"])')
//for i in range(10):
for it:= 1 to 10 do
  println(UTF8toAnsi(eg.evalStr('fake.name()')));
>>> Alma María José Montañez Dávila ...
```



The Faker constructor takes also a performance-related argument called use_weighting. It specifies whether to attempt to have the frequency of values match real-world frequencies and distribution shape (e.g. the English name Gary would be much more frequent than the name Welson). If use_weighting is False, then all items have an equal chance of being selected, and the selection process is much faster; the default is True.

The next line is a simple demonstration of Faker credit card:

```
println('creditcard#: '+eg.evalStr('fake.credit_card_number()')); //}
```

Faker also support for dummy hashes and uuids for SynDat:

```
#!/usr/bin/env python
from faker import Faker
faker = Faker()
print(f'md5: {faker.md5()}')
print(f'sha1: {faker.sha1()}')
print(f'sha256: {faker.sha256()}')
print(f'uuid4: {faker.uuid4()}')
```

In the end we close and free all the resources of objects, including stop-watcher sw and python frame apd:

```
except
  eg.raiseError;
  writeln(ExceptionToString(ExceptionType, ExceptionParam));
finally
  eg.Free;
  sw.Free;
  sw:= Nil;
  apd.position:= 100;
end;
```

You can also run the **Python Engine** script at runtime to get a `Faker()` object and if something went wrong you got a `raiseError Py exception.` `Eval()` function accepts a string argument and if the string argument is an expression then `eval()` will evaluate the expression as a `callback` with return (*faker.proxy.Faker*):

```
with TPythonEngine.Create(Nil) do begin
pythonhome:= PYHOME;
try
 loadDLL;
 Println('Faker Platform: '+
    EvalStr('__import__("faker").Faker()'));
except
 raiseError;
finally
 free;
end;
end;
```

CONCLUSION

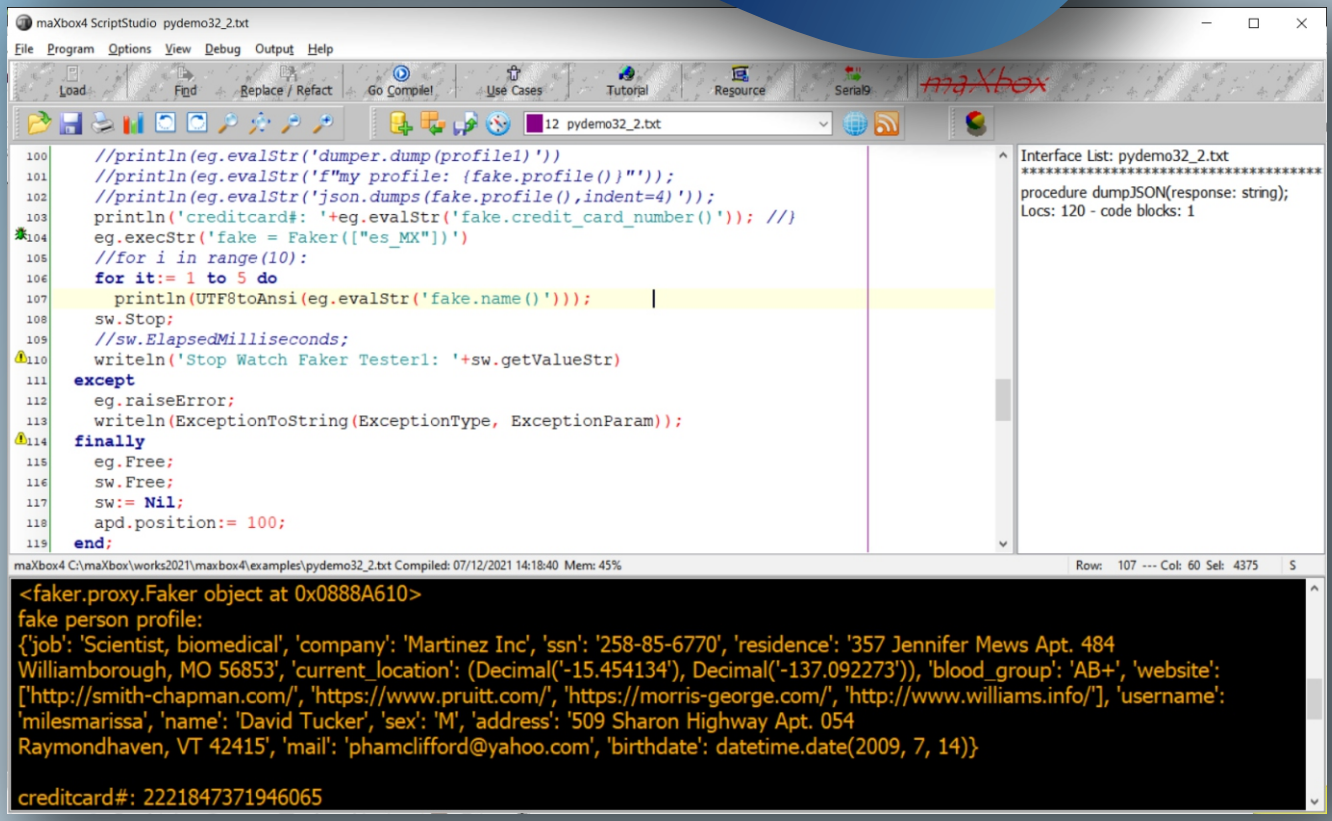
In this report, we used Python Faker to generate fake or synthetic data in Python and maXbox with measuring time behaviour.

Finally, synthetic datasets can minimize privacy concerns.
Attempts to anonymize data can be ineffective, as even if sensitive/identifying variables are removed from the `dataset`, other variables can act as identifiers when they are combined. This isn't an issue with synthetic data, as it was never based on a real person, or real event, in the first place.

A concept could mean, firms, institutes or simply users don't deal with real person data, they got an avatar which makes an relationship between a hash and a guid in a worldwide proxy block-chain (pb1).
A real person is protected behind the SynDat proxy with a guid record.

Python for .NET is also a package that gives Python programmers nearly seamless integration with the **.NET Common Language Runtime (CLR)** and provides a powerful application scripting tool for **.NET** developers and with **Delphi** or **Lazarus** just found that:

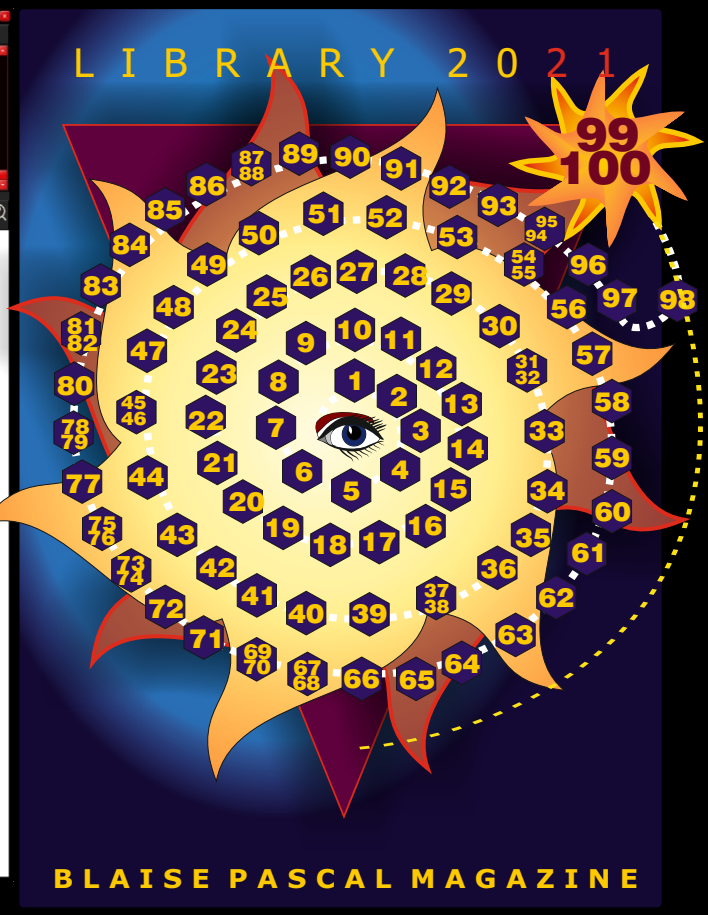
`https://i2.wp.com/blogs.embarcadero.com/wp-content/uploads/2021/07/demo01_Faker2-2809487.png?ssl=1`



SYNDAT TOPICS AND SCRIPT:

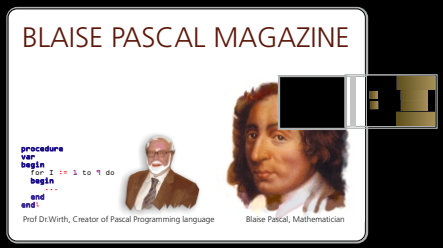
- https://pypi.org/project/Faker/
- https://www.kdnuggets.com/2021/11/easy-synthetic-data-python-faker.html
- http://www.softwareschule.ch/examples/pydemo32_2.txt
- https://www.unite.ai/what-is-synthetic-data/
- http://www.softwareschule.ch/examples/cheatsheetpython.pdf

```
**********************************************************************
Release Notes maXbox 4.7.6.10 II November 2021 mX476
**********************************************************************
Add 10 Units + 3 Tutorials
1441 unit uPSI_neuralgeneric.pas; CAI
1442 unit uPSI_neuralthread.pas; CAI
1443 unit uPSI_uSysTools; TuO
1444 unit upsi_neuralsets; mX4
1445 unit uPSI_uWinNT.pas mX4
1446 unit uPSI_URungeKutta4.pas ICS
1447 unit uPSI_UrlConIcs.pas ICS
1448 unit uPSI_OverbyteIcsUtils.pas ICS
1449 unit uPSI_Numedit2 mX4
1450 unit uPSI_PsAPI_3.pas mX4
Total of Function Calls: 35078
SHA1: of 4.7.6.10  D4B0A36E42E9E89642A140CCEE2B7CCDDE3D041A
CRC32: B8F2450F 30.6 MB (32,101,704 bytes)
```



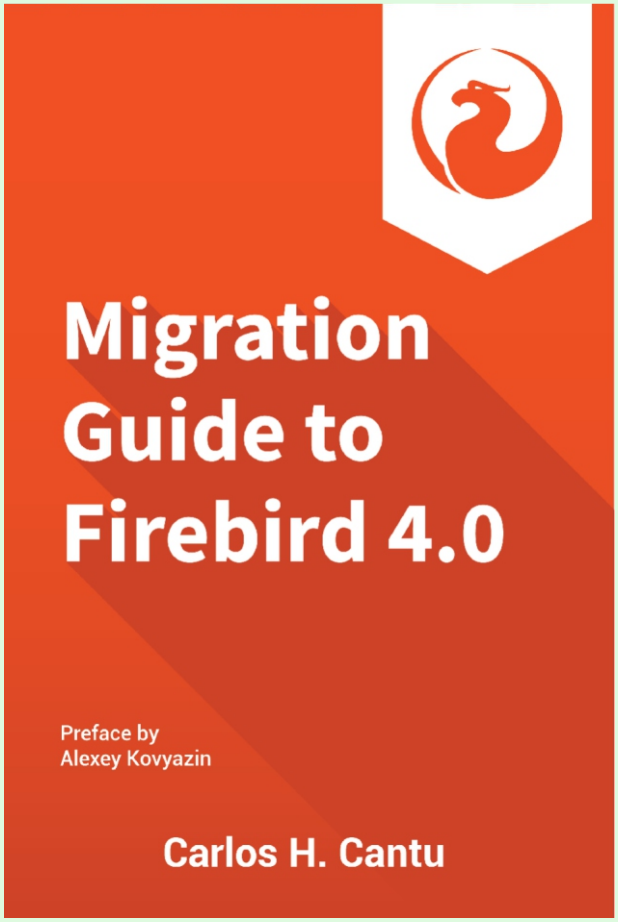
Blaise Library Program
+
USB Librarystick

Containing:
installer for Windows
Issues 1-100 / 5809 Pages
873 Articles / Code samples



Price: € 75 incl.Shipping

https://www.blaisepascalmagazine.eu/product/lib-stick
https://www.blaisepascalmagazine.eu/product/bundle-libstick-download-subscription/
https://www.blaisepascalmagazine.eu/product-category/special-offer/



ABSTRACT

Accompagnying the recent release of Firebird 4.0, a book about migrating to 4.0 is a welcome help for Firebird users who wish to use the latest version of the Firebird 4.0 engine. A review of the english translation of the book.

MIGRATION GUIDE TO FIREBIRD 4

First edition – 2021
Author: Carlos Henrique Cantu
Piracicaba – São Paulo – Brazil

Editing, translation, diagramming, finalization:
Carlos Henrique Cantu
Proofreading: Ann Harrison
Revision 1.20
The book is for sail at:

`https://www.firebirdnews.org/migration-guide-to-firebird-4/`

1 INTRODUCTION

As the title indicates, the book 'Migration guide to Firebird 4.0' by Carlos Henrique Cantu is meant for people who are already using Firebird: The book covers migration from version 3.0 of Firebird or earlier versions.
It is not meant as a beginner's guide to Firebird, nor is it intended to be a complete reference of Firebird.

The book gives insight in the isses you can (and most likely will) encounter when migrating existing databases and applications to Firebird. It also gives hints on how to solve or prevent the issues from occurring.

To this end, the book starts by repeating some basic firebird concepts: the various available architectures and their characteristics - important for choosing the right version to use. It mentions user-defined functions: these are deprecated in Firebird. They can still be used in Firebird, but they are no longer available or enabled by default.

2 INSTALLATION & MIGRATION

A first step in migrating to a new server version is obviously installing the new version, so this is covered to some extent for Linux and Windows: This chapter offers little surprises to seasoned Firebird users, as the procedure has not changed significantly.

The migration chapter is arguably the most important chapter of the book: it explains the need for a migration process, identifies the pitfalls that can occur during the migration and offers workarounds for some commonly found problems. It also recommends a replication scheme for migration of systems that must be available 24/7 but unfortunately, it fails to explain how to do this - earlier versions of Firebird do not have this functionality built-in, making this a non-trivial task which could really use an in-depth explanation.

A new installation needs to have some users present to be able to function, so some time is spent on explaning the new features regarding user management in Firebird: For users of firebird 3, this will offer few insights, but users of older versions of Firebird should read this chapter carefully, as the user management has changed significantly in version 3, so you need to be aware of it if you migrate to version 4.0.
SQL users are only one component of database security, and so the book spends some time on tips how to further secure your databases.

3 NEW FEATURES OF FIREBIRD 4.0

Good reasons for updating a database server are improved stability, speed and bug fixes.
Access to new features is an equally valid reason for migrating to a new version, so naturally the new features must be discussed in a book about migration to a new version.

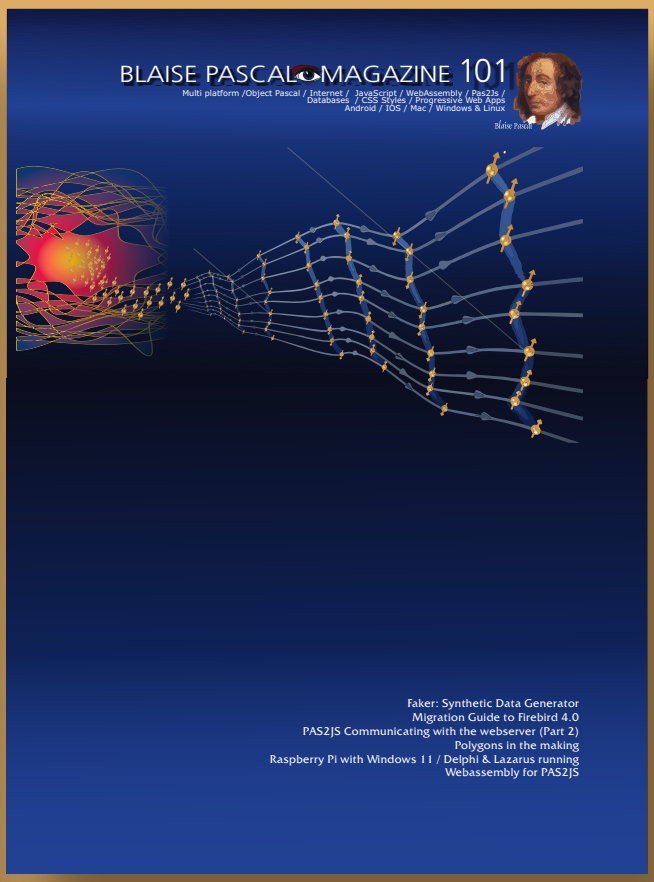
Firebird 4.0 - or more specifically, the client library that applications use when connecting to firebird - allows you to specify connection strings using an URL syntax, the book naturally explains how to construct these connection strings.

New in Firebird is how Firebird manages some aspects of transaction isolation. The book explains how the transaction isolation works and what changes were introduced in Firebird 4.0 - This chapter is mostly important for application programmers: the transaction isolation levels are usually controlled in application code.
The consequences of the new transaction isolation for garbage collection (and the automated sweep) are also explained.
Every new version of Firebird comes with new features in SQL, and version 4.0 is no different in this regard: new keywords are introduced as well as new data types: the new data types do not interfere with the migration process, but the new keywords can cause problems.

The new time zone capabilities of Firebird are treated in depth. Last but not least, with Firebird 4, firebird gets one-way replication capabilities: the required setup and parameters for database replication are treated in depth.

4 Conclusion

People that wish to migrate to Firebird 4.0 from earlier versions of Firebird will definitely find this book useful: In fact, people with older Firebird versions have more reason to buy this book, since it also discusses changes introduced in Firbeird 3.0. Written in an informal style, it is an easy read that will quickly get you up to speed with the latest version of Firebird.



USB LIB stick + 1 year subscription for only € 100

Blaise Library Program and USB Librarystick

Containing:
installer for Windows
Issues 1-100 / 5809 Pages
873 Articles / Code samples

D11 starter expert

PAS2JS PART 2

ABSTRACT

In a previous article we showed how to get started with pas2js, and how to compile a simple program that interacts with the **HTML** of the webpage. In this article, we show how to interact with an application server using **JSON-RPC**.

❶ INTRODUCTION

It is important to have a close look at the source code once you start acting Please read the article completely before working with it.

A webpage almost invariably communicates with services hosted on a webserver. This can go from downloading a simple file to exchanging data with an application server. As explained in the previous article about real-world programming with **PAS2JS** there are several communication protocols possible: **SOAP, REST, JSON-RPC.**

The communication can happen over **HTTP(s)** or using websockets. **Free Pascal** supports all of these with several frameworks – **FPC** can be used to write a **HTTP** server or **Websocket** server – or even both at the same time.

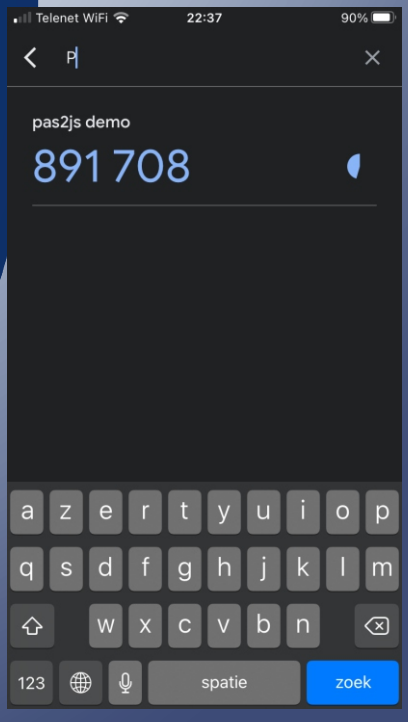
In this article, we'll explain how to use **JSON-RPC** on the server and in **Pas2JS.** The previous article laid the foundations for a login page, and we will now expand on this foundation to demonstrate how to let a **PAS2JS** program communicate with a server.

For this, we'll implement a Users service with 3 calls:

Login	The login call to let a user log in using a username and password.
Logout	The logout call.
CreateUser	A call to create a new user in the user database.

To make our application more secure, we'll also implement **2-factor authentication (2FA)** using the **Google Authenticator** application: **Free Pascal** has a unit that can generate a time-based token which can be used with the **Google Authenticator** application.

This means the login page presented in the previous article needs to be expanded, so we can ask the user for the **2FA** code. At the same time, we'll expand the **HTML** page a little, so it contains a menu bar in which we will add login and logout buttons as well as a place to show the user name.



❷ THE APPLICATION SERVER

To be able to create the application server, the **WebLaz** package package must be installed in the IDE. If this is not yet the case, you can install it in the same way as the **PAS2JSDSGN** package had to be installed for **Pas2JS** support, using the Packages - Install packages menu.

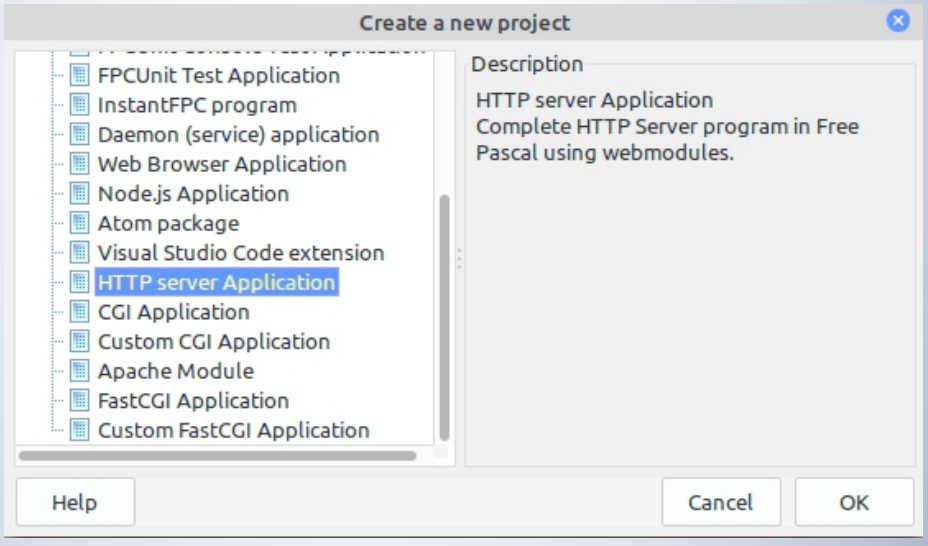


Figure 1: Choosing HTTP Server Application

Once the package is installed you can make several kinds of webserver applications: **CGI, FastCGI, standalone HTTP** server or an apache module.

The HTTP server application (*see figure 1 on page 2*) needs the least setup, and is easiest to debug, so we'll take that. For a production environment, it may be better to use **FastCGI** or even an apache module - but this can be easily changed later during development. Once you choose this project type, the new project wizard will then present you with some options, as seen in figure 2 on page 3.

The 'Port to listen for requests' is the **TCP/IP** port on which the server will listen. Any port can be entered, but take care that the port is not yet in use on your system, and that your user is allowed to use this port: on **Linux,** port numbers below 1024 are reserved for the root user.

If the 'Register location to serve files from' option is checked, the wizard will insert code to let the **HTTP** server automatically serve files. No special code will need to be written for that, so this is very convenient. In the 'Directory' edit box, the directory from which to serve files can be specified: Subdirectories will be handled, but the program will refuse to handle files outside that directory. For most cases, the base directory will need to be set correctly in code anyway.

In the 'location' edit you can enter the start of the **URL** the server needs to get to serve files. In the configuration as shown in figure 2 on page 3, the **URL**

```
http://localhost:3000/files/css/login.css
```

will be mapped to the following filename on disk:

```
/home/michael/logindemo/css/login.css
```

The 'Threaded' checkbox tells the wizard to generate a program that will use threads to serve requests in. Special care must be taken when handling database access when you use threads, so for the moment we'll leave this unchecked.
When you confirm the settings, the following program source code is generated:

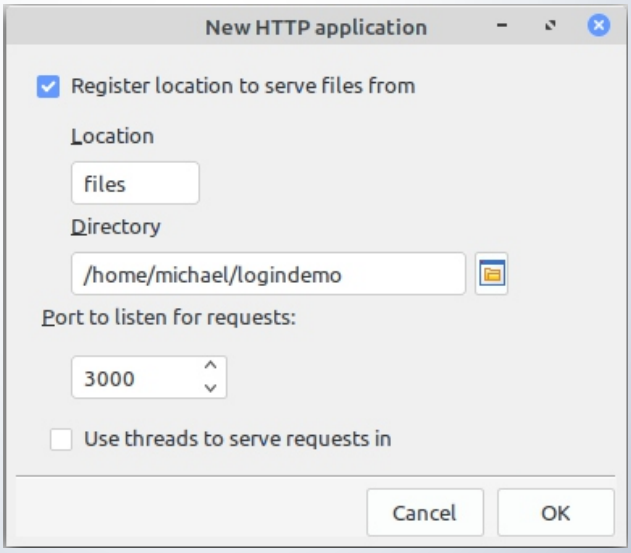


Figure 2: Options for a HTTP Server Application

```
program loginserver;

{$mode objfpc}{$H+}

uses sysutils, fpwebfile, fphttpapp, unit1;
begin
  RegisterFilelocation('files','/home/michael/logindemo/')
  Application.Title:='httpproject1';
  Application.Port:=3000;
  Application.Initialize;
  Application.Run;
end.
```

But we will change this to the following:

```
program loginserver;

{$mode objfpc}{$H+}

uses sysutils, fpwebfile, fpmimetypes, fphttpapp, unit1;
Var aDir : string;
begin
  MimeTypes.LoadKnownTypes;
  Application.Title:='Pas2JS demo server';
  Application.Port:=3000;
  Application.Initialize;
  if Application.HasOption('d','directory') then
     aDir:=Application.GetOptionValue('d','directory')
  else
     aDir:=ExtractFilePath(ParamStr(0))+'../webwidget/';
  TSimpleFileModule.BaseDir:=ExpandFileName(aDir);
  TSimpleFileModule.RegisterDefaultRoute;
  Application.Run;
end.
```

The reason for this change is 2-fold:

❶ The requirement to use the `'/files/'` prefix in all URLS to serve files is not very convenient. It would be better not to have to type this prefix. Instead, it is easier to let the **HTTP server** try to serve as a file any **URL** it does not recognize as special.
This is what the call to the `TSimpleFileModule.RegisterDefaultRoute` class method does: it will register the `TSimpleFileModule` class (*this class is a HTTP route handler made available by FPC*) as the default route handler of the server:
any non-recognized route will be treated as a file.

❷ We set the TSimpleFileModule.BaseDir class variable to the directory where the `TSimpleFileModule` must look for files. The location can be set with the `-d` command-line option. Because of the location of the login page client project, a default of `../webwidget/` relative to the server project directory is used.
NOTE that in the trunk version of **Lazarus,** the 'New HTTP application' wizard has been improved, so the above changes do not have to be made: the wizard now can be used to configure the `TSimpleFileModule` for you.

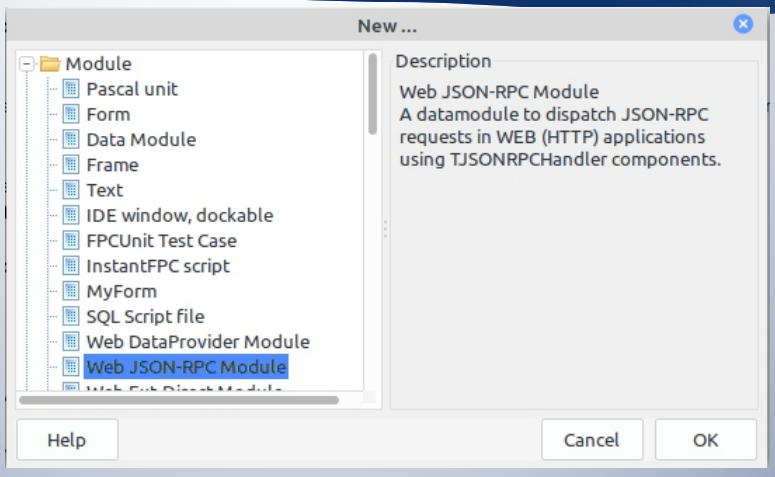


Figure 3: Creating a Web JSON-RPC module

❸ THE JSON-RPC SERVICE

The "New HTTP application" wizard has generated a first WebModule (a `TFPWebModule` descendent) in unit1. We don't need this webmodule, so we remove unit1 and the webmodule from the project and save the resulting project as 'loginserver'. Instead, we use the **File-New** menu dialog to create a Web **JSON-RPC** module (*see figure 3 on page 4*). This module is the basis for the RPC server.
The **RPC** server in **FPC** is currently implemented using 3 components:

❶ `TJSONRPCModule` This is a WebModule descendent that will serve **JSON-RPC** requests. You need at least 1 `TJSONRPCModule` in your application.

❷ `TJSONRPCHandler` This is a component which will handle exactly 1 **JSON-RPC** method. For each method you want to create in your **JSON-RPC** server, you must drop 1 `TJSONRPCHandler` component on a `TJSONRPCModule` descendent or a `Tdatamodule`.

❸ `TJSONRPCDispatcher` This is a component that can will dispatch a **JSON-RPC** call to the correct `TJSONRPCHandler` component. The `TJSONRPCModule` WebModule will automatically create an instance of `TJSONRPCDispatcher` if you didn't specify one in its Dispatcher property.

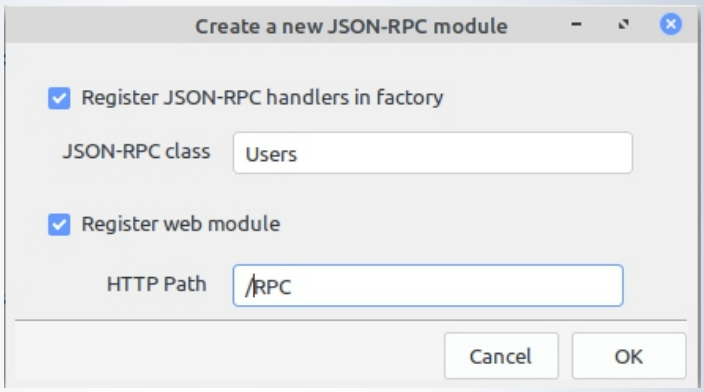


Figure 4: Options for the JSON-RPC module

Again, several options can (or must) be set to control the generated code, they are shown in figure 4 on page 5.
The **'Register JSON-RPC handlers in factory'** option can be left unchecked.

If checked, it will insert code that registers all `TJSONRPCHandler` components on the module in the **JSON-RPC** registry *(the factory pattern is used to find* **TJSONRPCHandler** *instances*). When doing so, it will use the **'JSON-RPC class'** as the class name for the **RPCJSON** registration: This is an extension in FPC which allows to have various 'classes' with methods in the JSON-RPC service.
The 'Register web module' will associate this web module with a **HTTP** route.
The value 'RPC' shown in figure 4 on page 6 means that the following **URL:**
`http://localhost:3000/RPC`
will be considered the entry point for the **JSON-RPC** service:
all requests to this URL will be handled by the **TJSONRPCModule.**

If you create multiple **TJSONRPCModule** modules, it is important you only register 1 of them as the handler for the **RPC** route, unless you want to create multiple routes for **RPC** calls.
For the other **TJSONRPCModule** modules, it is sufficient to use the 'Register **JSON-RPC** handlers in factory' option.
When you click **OK** in the new **JSON-RPC** module wizard and save the new unit as `dmRPC`, the following code will be generated:

```
unit dmRPC;

{$mode ObjFPC}{$H+}

interface

uses
  Classes, SysUtils, HTTPDefs, websession, fpHTTP, fpWeb,
  fpjason, fpjsonrpc, webjsonrpc;
type
  { TUsersModule}
  TUserModule = class(TUsersModule)
  private
  public
end;
  implementation
  initialization
     RegisterHTTPModule('RPC', TUsersModule);
     JSONRPCHandlerManager.RegisterDatamodule(TUsersModule, 'RPC', );
end.
```

In your code is `TJasonrpcmodule` *has been replaced by* `TUserModule`. There was a bug which has already been fixed in the trunk version of **Lazarus.**
Start by renaming the webmodule to `TUsersModule`, and then the `initialization` section code must be changed to the following:

```
initialization
  RegisterHTTPModule('RPC', TUsersModule);
end.
```

To add methods to our **JSON-RPC** server, we must drop a component on the datamodule: one component per method.

The `TJSONRPCHandler` class has the following important properties:

- **Name** The component name serves also as the method name. The most recent version of the component in **FPC** allows you to specify an alternate name.
- **Options** There are several options that can be set here:
  - `jroCheckParams` The type and number of incoming parameters is checked against the parameter definitions in **ParamDefs.**
  - `jroObjectParams` The parameters must be specified as a **JSON** object.
  - `jroArrayParams` The parameters must be specified as a **JSON** array.
  - `jroIgnoreExtraFields` If the call has extra parameters on top of the parameter definitions in **ParamDefs** they are ignored.
- **ParamDefs** This collection property serves 2 purposes: It is used when generating the description of the full **JSON-RPC API.** This collection has one item for each expected parameter to the method, in the order that they should be passed to the method. Every item in the collection has 3 properties: **Name, DataType** (one of the valid **JSON** types) and **Required.**

And the following event handlers exist:

- `OnExecute` This is the most important event handler: this event handler is called whenthe **JSON-RPC** method must be executed. It will get passed the parameters received from the client, and must return a **JSONData** value that is the result parameter.
- `BeforeExecute` This is an event handler that is called before actually executing the method. Here you can implement authentication or logging.
- `AfterExecute` This is an event handler that is called after the method was executed.
- `OnParamError` This event handler is called when the `jroCheckParams` option is specified and there is a parameter mismatch in the received parameters.

For most applications, it is sufficient to set the `OnExecute` event handler.
For our application, we need to implement a login call and a call to create a new user. For this, we'll store the allowed users in the database. For simplicity we will use a **Firebird** database, with the following definition for the users table:

```
CREATE SEQUENCE GEN_USERS;
CREATE TABLE USERS (
    U_ID BIGINT NOT NULL ,
    U_NAME VARCHAR(50) NOT NULL,
    U_PASSWORD VARCHAR(50) NOT NULL,
    U_2FASEED VARCHAR(32) NOT NULL,
    CONSTRAINT PK_USERS PRIMARY KEY (U_ID)
);
CREATE UNIQUE INDEX UDX_USERS ON USERS(U_NAME);

set term ^;

CREATE TRIGGER TR_INSERTID FOR USERS
BEFORE INSERT
AS
BEGIN
    IF (NEW.U_ID IS NULL) THEN
    NEW.U_ID=GEN_ID(GEN_USERS,1);
END^
```

The `U_2FASEED` field serves to store a shared secret for 2-factor authentication. The login call needs 3 parameters: username, password and the 2-factor authentication code. These can be entered in the **ParamDefs** property, as shown in figure 5 on page 7. Now we can actually implement the **Login** call. For this we assign the `OnExecute` handler, and enter the following code:

```
procedure TUsersModule.LoginExecute(Sender: TObject;
const Params: TJSONData;
out Res: TJSONData);
Var A : TJSONArray absolute Params;
    aUserName,aPassword : String;
    aTwofactorCode : Integer; OK : Boolean;
begin
   aUserName:=A.Strings[0];
   aPassword:=A.Strings[1];
   aTwoFactorCode:=A.Integers[2];
   OK:=CheckUser(aUserName,aPassword,aTwoFactorCode);
   Res:=TJSONBoolean.Create(OK);
   if OK then
      Session.Variables['User']:=aUserName;
end;
```

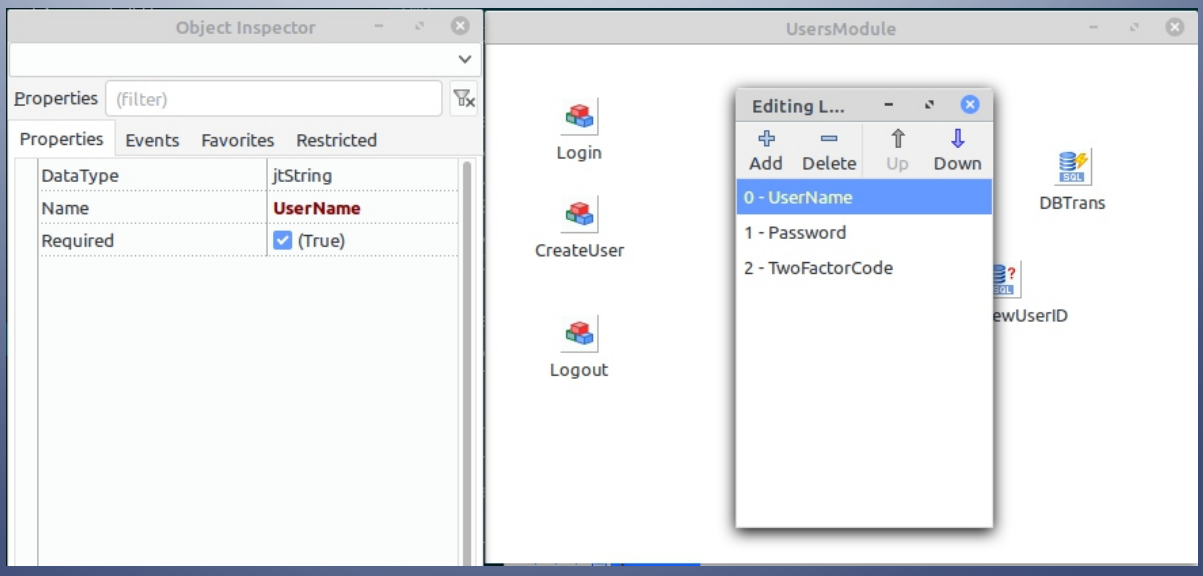


Figure 5: Parameters for the login call

The first 3 lines simply save the values of the parameters, and in the 4th line the call to `CheckUser` will actually check the passed parameters with the contents of the database.

If the call returns True, we store the username in the browser session: this way we can verify whether the browser user is authenticated or not in future calls.

The `CheckUser` method is actually pretty standard code to run a query and compare the contents of the password with the password stored in the database: The routine starts by connecting to the database, using the `ConnectDB` call. The details of this method we will not describe here; the interested user can consult the source code.

```
function TUsersModule.CheckUser(const aName, aPassword: String; ACode: Integer): Boolean;
Var DBPassword, Secret: string;
begin
  ConnectDB;
  With QGetUser do
    begin
      ParamByName('NAME').AsString:=aName;
      Open;
    try
    Result:=Not IsEmpty;
      if Result then
       begin
          DBPassword:=FieldByName('U_PASSWORD').AsString;
          Secret:=FieldByName('U_2FASEED').AsString;
          Result:= (aPassWord=DBPassword);
          If Result Then
                Result:=Check2FA(Secret,aCode);
       end;
    finally
      Close;
    end;
  end;
end;
```

The `QGetUser` is a **TSQLQuery** component which was dropped on the RPCModule. We enter the following SQL command in its SQL property:

```
SELECT
   U_ID, U_PASSWORD, U_2FASEED
FROM
   USERS
WHERE
   (U_NAME=:NAME)
```

If the query does not return a result, we know the username is not known and authentication should fail.
If the query returns a result, we verify the password in code.
To be really safe, it would of course be better to save the password in hashed form and compare the hashed form of the incoming password with the hash stored in the database.

If the username was correct and the password matched the password stored in the database, the **2FA** shared secret stored in the database is used to check the **2-factor authentication** code:

```
function TUsersModule.Check2FA(const aSeed : String; aCode : Integer) : Boolean;
Var O : Integer;
begin
  Result:=TOTPValidate(aSeed,aCode,1,O);
end;
```

The `TOTPValidate` function is implemented in the onetimepass unit, part of **FPC.**

```
function TOTPValidate(const aSecret: AnsiString; const Token: LongInt;
                      const WindowSize: LongInt;
                      var Counter: LongInt): Boolean;
```

It requires a secret, the token to verify code, a maximum allowed deviation (*the windowsize, measured in seconds*). It will return a counter – the counter can be used to implement a counter-based verification token, but we will not use that and stick to a time-based token. The logout call is very simple. It does not need any parameters at all, and the code is quite simple.

```
procedure TUsersModule.LogoutExecute(Sender: Tobject;const Params: TJSONData; out Res: TJSONData);
begin
  Session.Variables['User']:='';
  Res:=TJSONNull.Create;
end;
```

It is important to always set the res result, even if it is `Nil`:
failing to do so will lead to unpredictable behaviour.
To create a user, we require a username and password, and simply insert a record in the database. This can be done again with very little code. We start by verifying whether the current user is allowed to do so.
For the current demonstration application, we check that the user is the **Admin** user, and we raise an exception when the user is not the administrator user. The exception will be caught by **FPC's JSON-RPC** implementation, and translated to a valid **JSON-RPC** error response:

```
procedure TUsersModule.CreateUserExecute(Sender: Tobject; const Params: TJSONData; out Res: TJSONData);
Var
   A : TJSONArray absolute Params;
   aUserName,aPassword : String;
   aID : Int64;
begin
   if Session.Variables['User']<>'Admin' then
     Raise Exception.Create('Only admins can create users');
   aUserName:=A.Strings[0];
   aPassword:=A.Strings[1];
   aID:=DoCreateUser(aUserName,aPassword);
   Res:=TJSONInt64Number.Create(aID);
end;
```

After collecting the username and password from the passed parameters, the `DoCreateUser` method is called to actually create the user in the database.
The response of the `DoCreateUser` call is the **ID** of the new user record in the database;
For **Firebird** (or other databases that support sequences), this **ID** can be fetched separately, this is implemented in the `GetNewUserID` method using a simple query.

```
function TUsersModule.DoCreateUser(const aName, aPassword: String): Int64;
Var
  aSeed : String;
begin
  aSeed:=TOTPSharedSecret();
  Result:=GetNewUserID;
  ConnectDB;

  With QInsertUser do
  begin
    ParamByName('ID').AsLargeInt:=Result;
    ParamByName('NAME').AsString:=aName;
    ParamByName('SEED').AsString:=aSeed;
    ExecSQL;
  end;
end;
```

The `TOTPSharedSecret` call (*also implemented in the onetimepass unit*) creates a new(*random*) shared secret usable in the **Google Authenticator.** It is stored in the database together with the new user record.

4 2FA AND THE GOOGLE AUTHENTICATOR APP

The **Google authenticator** works using a shared key: the application that wishes to authenticate a user using **2FA** generates a shared secret and stores this somewhere, associated with the user.

The user registers this shared key in the **google authenticator**. The app uses this secret to generate a secret code every 30 seconds. When asked for it, the user enters this code in the web application. The web application server also generates the secret code using the shared key associated with the user, and compares it with the code given by the user: if they match, it confirms the identity of the user.

Obviously, the shared secret for **2-Factor authorization** must be communicated by some safe means to the user when the new user record is inserted in the database. The application as it is now does not provide any method to communicate this secret – it would lead too far to discuss that. Converting it to a **QR** code is one way, sending the code by text or some other means is another.

To use the shared secret, the **Google Authenticator** application must be installed on a device (*smartphone, tablet*) owned by the user: this can be an **IOS or Android** device.
In the **Google authenticator** app, the user must add a new key using the 'Add' button and selecting either 'Scan a QR code' or 'Enter key', after which a description and the shared secret can be entered as in *figure 6 on article page 11*.
When it is time to authenticate, the application asks for the authentication code, and the user has 30 seconds to enter the code displayed in the **Google authenticator** app in the website, *see figure 7 on article page 11*, after 30 seconds, a new code is generated.

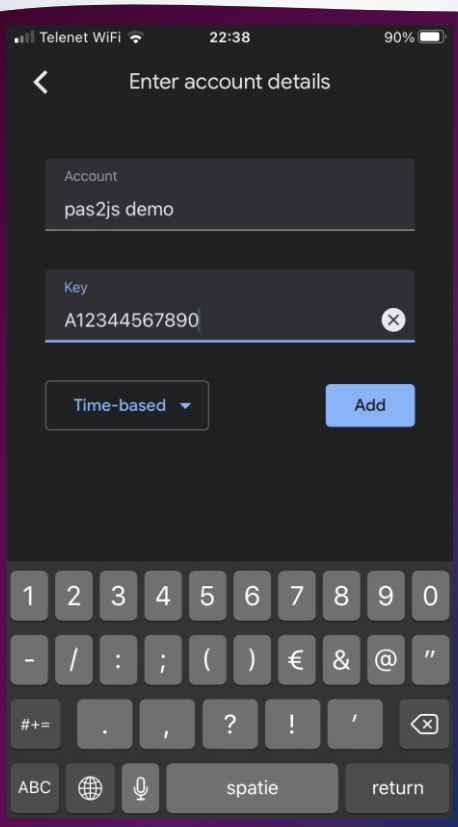


Figure 6: Adding the shared secret

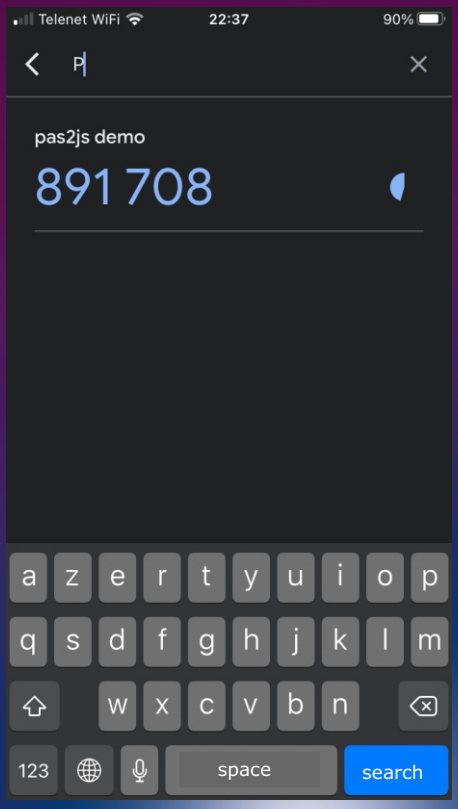


Figure 7: Adding the shared secret

5 MODIFYING THE CLIENT

Now that we have the server programmed, it is time to enhance the client application. In the previous article 2 applications were presented: one programmed with plain **HTML** classes, one with **WebWidget** components.
Here we will only enhance the application programmed with **WebWidget** components, but the sample client application written using plain **HTML** classes can be adapted in much the same way. We'll start by adding a menu (*a navbar in web parlance, using the <nav> tag*). The **HTML** for the navbar can be found in the code accompanying this article, but the resulting nav bar is shown in *figure 8 on article page 11*.

The navbar **CSS** in Bulma is quite simple, is responsive, and features a hamburger menu – the three little lines that appear hen the **CSS** hides the menu on small screens. When clicking the hamburger menu, the menu itself must be shown or hidden in code.

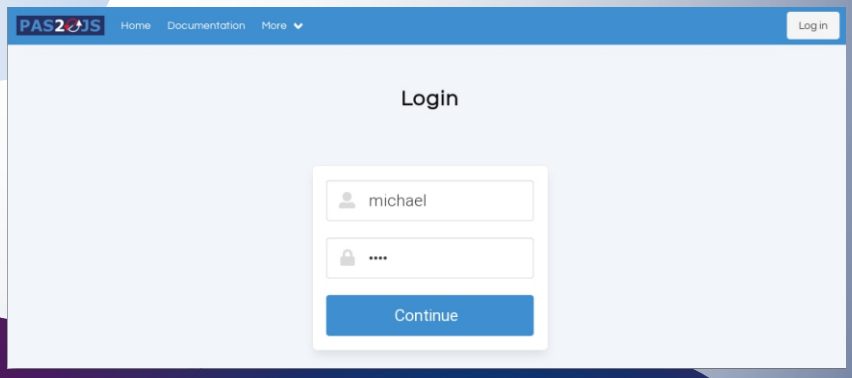
Contrary to **CSS** frameworks such as **Bootstrap, Bulma** does not offer some standard **Javascript** file to perform this task, but the task is programmed easily enough: We assign an ID to the hamburger menu tag, and to the menu itself. This allows us to create webwidgets that reference these tags, and we can attach a `OnClick` handler to the hamburger `<div>` tag. As shown in the previous article, this is done in the BindElements routine:

```
divMenuHamburger:=TTagWidget.Create(Self);
divMenuHamburger.elementID:='navbar-burger';
divMenuHamburger.Refresh;
divMenuHamburger.OnClick:=@ClickNavBar;
divMenu:=TTagWidget.Create(Self);
divMenu.elementID:='navbar-main';
divMenu.Refresh;
```

The **ClickNavBar** method is then simply:

```
procedure TMyApplication.ClickNavBar(Sender: TObject; Event: TJSEvent);
  begin
    if Pos('is-active',divMenu.Classes)<>0 then
      DivMenu.RemoveClasses('is-active')
    else
      DivMenu.AddClasses('is-active');
end;
```

Figure 8: The navigation bar in a large screen



The **AddClasses** and **RemoveClasses** methods add or remove **CSS** classes to the **HTML** tag of a widget.
Applying the CSS class is-active shows the menu, if it is absent, the menu is hidden.
The result can be seen in *figure 9 on article page 12.*
In the **navbar,** we add some menu items (*not functional at this time*), and also buttons to log in or log out and a small section to display the user name once the user is logged in.
The log in button will display the login dialog, and the log out button will log out the user and then displays the login dialog.
For each of these elements, a **TTagWidget** is made and associated with the corresponding **HTML** tag using the **ID** in the **BindElements** call:

```
divMenuLogin:=TTagWidget.Create(Self);
divMenuLogin.elementID:='mnuLogin';
divMenuLogin.OnClick:=@DoLoginMenuClick;
divMenuLogin.Refresh;
divMenuLogout:=TTagWidget.Create(Self);
divMenuLogout.elementID:='mnuLogout';
divMenuLogout.Refresh;
divMenuLogout.OnClick:=@DoLogoutClick;
```

```
procedure TMyApplication.DoLoginMenuClick(Sender: TObject; Event: TJSEvent);
begin
  divDlgLogin.RemoveClasses('is-hidden');
  ShowLogin('');
end;

procedure TMyApplication.DoLogoutClick(Sender: TObject;  Event: TJSEvent);
begin
  ShowLogout;
  DoLogout;
end;
```

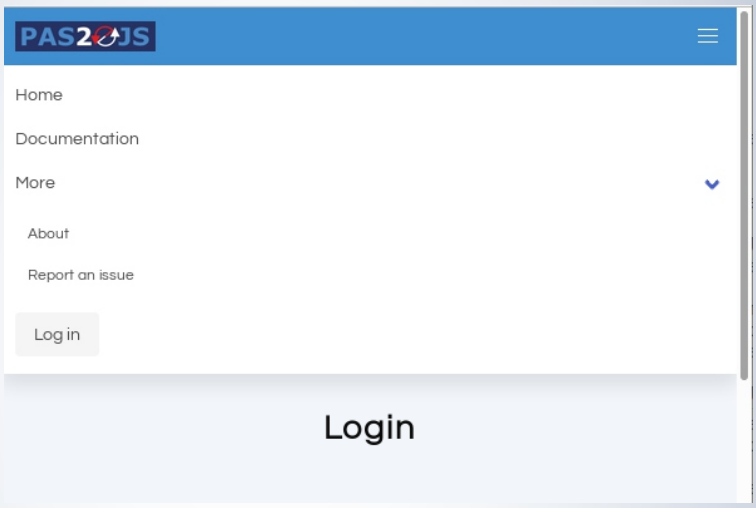


Figure 9: The navigation bar in a small screen

The `DoLogout` method will do the actual logout call to the server.
The `showLogin` and `ShowLogout` methods simply hide or show various **HTML** tags:

```
procedure TMyApplication.ShowLogout;
begin
  divDlgLogin.RemoveClasses('is-hidden');
  ShowLogin('');
  divMenuUser.AddClasses('is-hidden');
  divMenuLogin.RemoveClasses('is-hidden');
 divMenuLogout.AddClasses('is-hidden');
end;
```

The is-hidden **CSS** class is provided by **Bulma** and will hide the element to which it is applied.
The `ShowLogin` does a little more work. Because we wish to have **2-Factor Authorization**, the login dialog is split in 2 parts:
the **first part** asks for the user **name** and **password,**
and **the second part** asks for the **2FA** code.
Each part is contained in a **Bulma** "box" tag, having an **ID** of **div2FA** and **Login.**
Again each tag will be represented by a `webwidget` and bound to the **HTML** tag in the `BindElements` method. The `ShowLogin` method shows the Login box, but hides the `2FA` box.
Additionally, `ShowLogin` will be called after a failed login attempt, to allow the user to start over: in that case an error message is passed in the `aError` parameter. If it is non-empty, an `Error` div is shown or hidden beneath the username entry.

```
procedure TMyApplication.ShowLogin(const aError : String);
begin
  div2FA.AddClasses('is-hidden');
  divLogin.RemoveClasses('is-hidden');
  if aError<>'' then
    begin
      divError.RemoveClasses('is-hidden');
      divError.TextContent:=aError;
    end
  else
    begin
      divError.AddClasses('is-hidden');
      divError.TextContent:='';
    end;
end;
```

As can be seen in *figure 8 on article page 12*, the login button from the first version of the application has been changed so it displays the `Continue` text. When pressed, it hides the login box, and displays the **2FA** Box:

```
procedure TMyApplication.doContinueClick(sender : Tobject; event : TJSEvent);
begin
  divLogin.AddClasses('is-hidden');
  div2FA.RemoveClasses('is-hidden');
end;
```

The result of this is that the user sees the following part of the login dialog, where he must ented the **2FA** authentication code, as shown in *figure 10 on article page 14*. After that, the login actually can be performed.

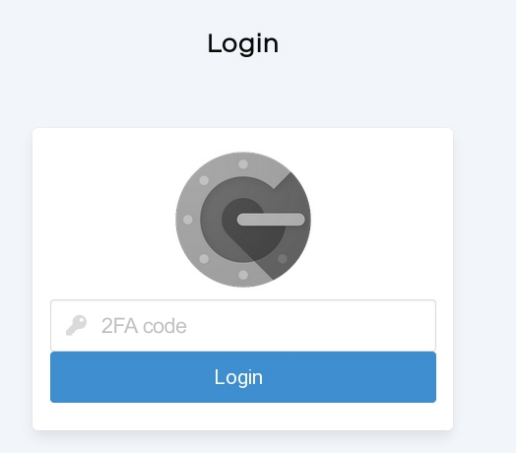


Figure 10: The 2-Factor authentication code

6 ACTUALLY TALKING TO THE SERVER: THE RPC CLIENT

Our server can handle the **JSON-RPC** protocol.
By definition, this means of course we must send **JSON** to the server. To send JSON to a server is easily done in a browser: the required **JSON** is easily constructed, and the `XMLHTTPRequest` class or the `Fetch` call can be used for sending it to the server and handling the response.
JQuery offers a ajax method. All these can be used in **Pas2JS.**
But **Pas2JS.** comes with a `TRPCClient` class which is geared specially towards **JSONRPC:** it will automatically create the correct envelope, it assigns the required keys such as **id** and **jsonrpc,** and takes care of error responses:
all kind of things that one would expect. But it also offers batching: you can batch calls explicitly, or let the client perform automatic batching; calls are batched, and after a configurable time, the batch is sent to the server.
This can be used to improve performance:
by sending several method calls in 1 **HTTP** request, the time spent on network communication is reduced. This class is defined as follows, with only the relevant methods and properties:

```
TRPCClient = class(TComponent)
Public
  Constructor Create(aOwner : TComponent); override;
  Destructor Destroy; override;
  Function CreateRequestParamsBuilder : TRPCRequestParamsBuilder;
  Function ExecuteRequest(const aClassName,aMethodName : String; aParams : TJSArray;
                          aOnSuccess : TRPCResultCallBack = Nil;
                          aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
  Function ExecuteRequest(const aClassName,aMethodName : String; aParams : TJSObject;
                          aOnSuccess : TRPCResultCallBack = Nil;
                          aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
  Procedure CloseBatch;
Published
  Property URL : String;
  Property Options : TRPCoptions;
  Property BatchTimeout: Integer;
  Property JSONRPCversion : String;
  Property CustomHeaders : TStrings;
  Property OnConfigRequest : TRPCConfigRequest;
  Property OnCustomHeaders : TRPCHeadersRequest;
  Property OnUnexpectedError : TRPCUnexpectedErrorCallback;
end;
```

The 2 `ExecuteRequest` methods will be treated below. The **URL** property is the URL for the RPC server. For our application, this will be something like /RPC
Or, equivalently
`http://localhost:3000/RPC`

The **Options** property is a combination of the following values:
`roParamsAsObject` create a parameter builder (*see below*) that creates the parameters as an object.

- `roFullMethodName`
  Combine classname and method name into a single name, using a dot as the separator. The effect of this option is that the name of the RPC method becomes '`classname.methodname`'. The default is to send the classname in a separate 'class' key.
- `roUseBatch`
  ExecuteRequest will not send the **JSON-RPC** request to the server at once. Instead, calls are batched and sent when `CloseBatch` is called.
- `roAutoBatch`
  If `roUseBatch` is specified together with this value, the first call to `ExecuteRequest` that starts a batch sets a timer: when the timer expires, `CloseBatch` is called automatically.
- `roForceArray`
  In case only 1 **RPC** method call is sent, force use of an array. By default, if only a single method is executed, only the object describing the call is sent. With this option enabled, the object is wrapped in an array. You can use this option if the **JSON-RPC** server is only capable of receiving arrays.

To execute methods on the server, the following 2 calls are important:

```
// Execute a request. Params can be passed as object or array
Function ExecuteRequest(const aClassName,aMethodName : String;
  aParams : TJSArray;
  aOnSuccess : TRPCResultCallBack = Nil;
  aOnFailure: TRPCFailureCallBack = nil) : NativeInt;

Function ExecuteRequest(const aClassName,aMethodName : String;
  aParams : TJSObject;
  aOnSuccess : TRPCResultCallBack = Nil;
  aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
```

The **aClassName, aMethodName** parameters are used to select the method to execute: they map directly to the classname, methodname used on the server. The parameters to the calls can be passed as an **Javascript** array or object.

The last 2 parameters are callbacks (*event handlers*) which will be called in case of success or failure to execute the call. Since every request to the server is asynchronous, a callback mechanism is needed (*a second mechanism using Javascript promises is in the works*).
The result of these functions is the id of the method call in the **JSON-RPC** protocol.
For example, to execute the **Login** call, we could create the following code:

```
var
  Params : TJSArray;
begin
  Params:=TJSArray.New('Michael','Secret',123);
  RPCClient.ExecuteRequest('Users','Login',Params,@DoOK,@DoFail);
end;
```

In this code, we assume that the **RPCClient** is set up appropriately elsewhere. The **DoOK** and DoFail methods could look like this:

```
procedure DoOK(aResult: JSValue);
begin
  if not Boolean(aResult) then
    ShowLogin('Invalid combination of username/password')
else
  StartLogin(aUser);
end;

procedure DoFail(Sender: TObject; const aError: TRPCError);
begin
  ShowLogin('Error during login: '+aError.Message);
end;
```

The **aUser** variable contains the user name. Note that the **DoOK** call uses a **JSValue** parameter (equivalent to a **Variant**).
In the above code you can see the use of the **ShowLogin** method introduced earlier, and also how an error is reported by the **TRPCClient** class: a **TRPCError** record is used.

```
TRPCError = record
ID : NativeInt;
Code : NativeInt;
Message : String;
ErrorClass : String;
end;
```

This structure is used both for server errors as for communication errors. The meaning of the fields should be clear from their names: the **Code** and **Message** are taken from the **JSON-RPC** protocol. In case of **HTTP** protocol errors, they will contain the **HTTP** status code and text. If an exception class name is available, it is reported in the **ErrorClass** field. **ID** contains the **ID** from the call.

The `StartLogin` method simply displays the user name in the `navbar` and hides the login dialog and login button, and shows the logout button instead, a matter of adding or removing the is-hidden **CSS** class:

```
procedure TMyApplication.StartLogin(Const aUser : String);
begin
divdlgLogin.AddClasses('is-hidden');
divMenuLogout.RemoveClasses('is-hidden');
divMenuLogin.AddClasses('is-hidden');
divMenuUser.RemoveClasses('is-hidden');
divLblUser.TextContent:=aUser;
FLoggedInUser:=aUser;
end;
```

7 USING A SERVICE CLASS

As can be seen from the above code sample, a login call is not difficult to code, but this method does have some drawbacks:

- It is not type safe. Both parameters and return value are not checked.
- If you must do the same call in different places in the application, it would be better to have a single, typed call that can be reused.

These 2 problems can be solved easily. The `TRPCCustomService` class (part of the fprpcclient unit) is meant to act as a base class with which a proxy class can be constructed for the 'Classes' defined by the **JSON-RPC** server. It introduces some auxiliary methods that help in building the parameters for the `ExecuteRequest` class. For our login **RPC** server, this proxy could be defined as follows:

```
TUserService = Class(TRPCCustomService)
  Protected
    Function RPCClassName : string; override;
  Public
    Function Login(Const aUserName,aPassword : String;
      aCode : Integer;
      aOnSuccess : TBooleanResultHandler = Nil;
      aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
    Function Logout(aOnSuccess : TEmptyResultHandler = Nil;
      aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
    Function CreateUser(Const aUserName,aPassword : String;
      aOnSuccess : TNativeIntResultHandler = Nil;
      aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
end;
```

As you can see, the public methods here mimic exactly the definition of the methods defined in out **JSON-RPC** server. The success handler is also typed: instead of a **JSValue** return, a **Boolean** return is expected. The `RPCClassName` must return the name of the class on the server:

```
function TUserService.RPCClassName: string;
begin
  Result:='Users';
end;
```

And the implementation of the login call can be done as follows:

```
function TUserService.Login(const aUserName, aPassword: String;
  aCode: Integer;
  aOnSuccess: TBooleanResultHandler;
  aOnFailure: TRPCFailureCallBack): NativeInt;

Procedure DoSuccess(Sender : TObject; const aResult : JSValue);
begin
  If Assigned(aOnSuccess) then
    aOnSuccess(Boolean(aResult));
end;
Var
  _Params : JSValue;
begin
  StartParams;
  AddParam('UserName',aUserName);
  AddParam('Password',aPassword);
  AddParam('Code',aCode);
  _Params:=EndParams;
  Result:=ExecuteRequest(RPCClassName,'Login',_Params,
  @DoSuccess,aOnFailure);
end;
```

The **StartParams** and **EndParams** methods of the **TRPCCustomService** class respectively create an instance of the **TRPCRequestParamsBuilder** class, and return the final parameters for use in the **ExecuteRequest** call. The **TRPCRequestParamsBuilder** class will create the parameters for the call as required by the settings of the **TRPCClient** class: as a **JSON** array or a **JSON** object.

The **AddParam** method of the **TRPCCustomService** class adds a parameter to the list of parameters: an overloaded version of this call exists for every supported **JSON** type.
Finally, **ExecuteRequest** is called and the success callback is re-routed through a local method, which will typecast then result to the appropriate type for the Login success callback (*in this case, a boolean*).
The result of all this is that now you can instantiate an instance of **TUserService** and do:

```
procedure TMyApplication.doServerLogin(const aUser,aPassword: String; aCode : Integer);
  procedure DoOK(aResult: Boolean);
  begin
    if not aResult then
      ShowLogin('Invalid combination of username/password')
    else
      StartLogin(aUser);
  end;

  procedure DoFail(Sender: TObject; const aError: TRPCError);
  begin
    ShowLogin('Error during login: '+aError.Message);
  end;

begin
  FUserService.Login(aUser,aPassword,aCode,@DoOK,@DoFail);
end;
```

The `FUserService` variable is an instance of the `TUserService` class. This code is reusable and type-safe.
The setup of the service and **RPC** client is very simple:

```
procedure TMyApplication.SetupServices;
begin
  FRPCClient:=TRPCClient.Create(Self);
  FRPCClient.URL:='/RPC';
  FUserService:=TUserService.Create(Self);
  FUserService.RPCClient:=FRPCClient;
end;
```

This code can be called for example in the constructor of the application object.
It is possible to code the `TUserService` class manually. But even this is not necessary:
The **FPC JSON-FPC** server can generate a description of the available services and method calls in **JSON** format: an extension of the **Ext.Direct** format used by **ExtJS.**

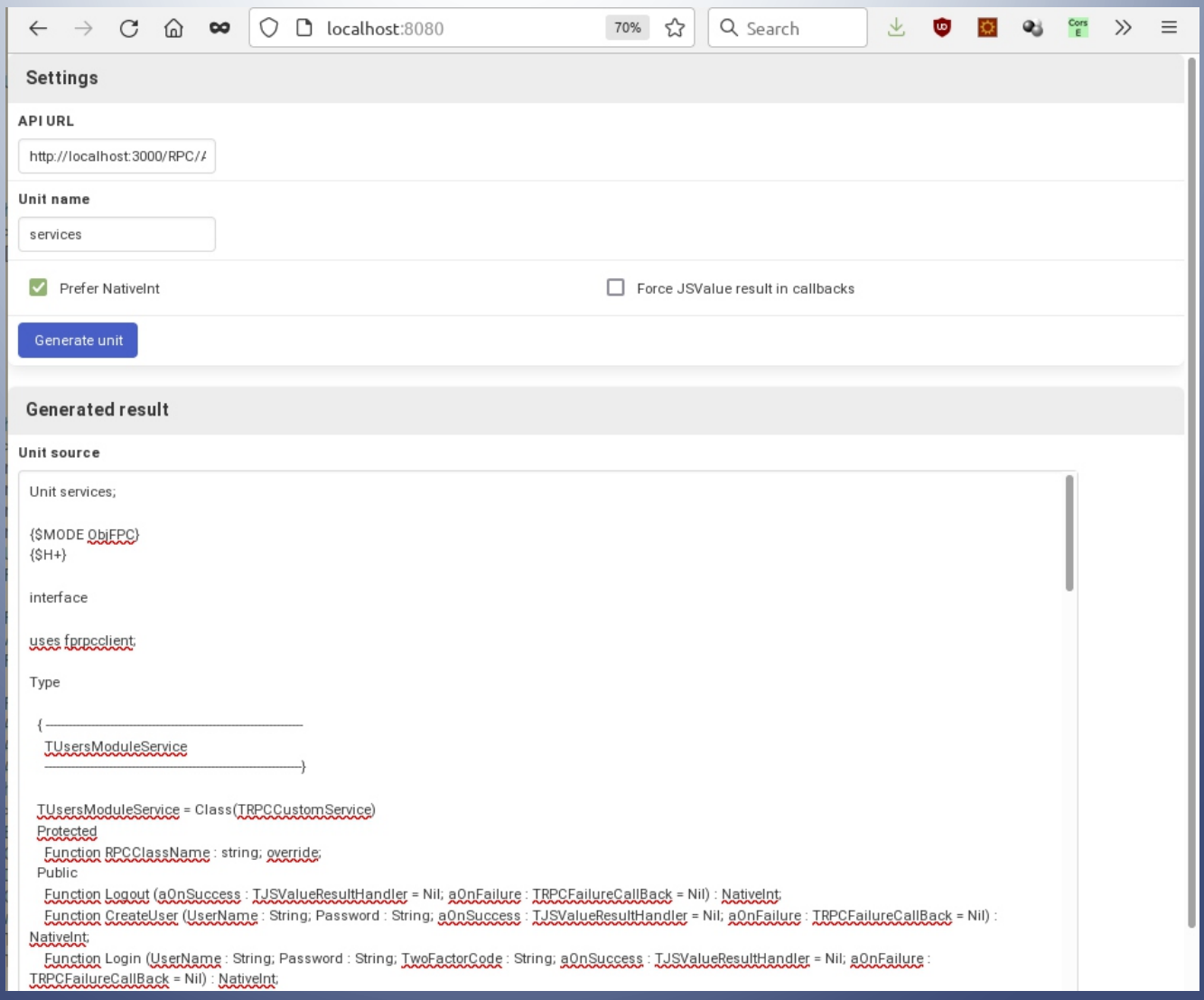


Figure 11: Automatically generating service code

Using the class `TAPIClientCodeGen` from the fprpccodegen unit (available in native **FPC** and in **pas2js**) the **JSON** description can be consumed and a unit with the above service code can be automatically generated. The generated unit will contain a service class for every class exposed by the **FPC JSON-RPC** server.

The **pas2js** distribution contains a demo project (apiclient) that uses this unit and allows you to generate the service classes exposed by a server, 100% automatically. All that is required is the URL where the **FPC JSON-RPC** server is listening for requests.
It is shown in *figure 11 on article page 19.*

Better yet, the trunk version of the **FPC JSON-RPC** server code can generate this code automatically, you can get it by entering the following **URL** in the browser:

```
http://localhost:3000/RPC/API?format=pascal&unitname=services
```

This way, your service description can be regenerated at any moment, and will always reflect exactly what the server is expecting as input and what data it is returning.
Since the **JSON RPC** server only supports **JSON** types, the generated code can only use the generic **JSON** types when generating code. An extension is planned where type hints can be given and for example a record type can be specified instead of a generic `TJSONObject` class, or a `TDateTime` instead of a `string`.

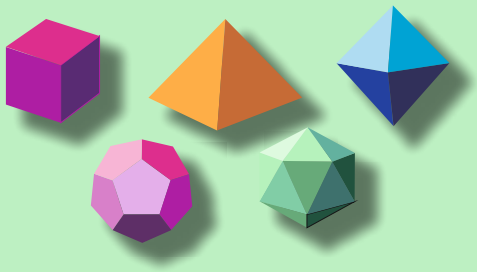
8 CONCLUSION

In this article we have shown how to construct a **RPC** server using a click-and-point mechanism. We've also shown how to call the **RPC** server and how to generate a service description.

The **GUI** of our application has been expanded, and when you look at the BindElements method, you'll see that this has become quite large. In the next article, we'll show how to generate this code automatically, and how to load the **HTML** for the dialogs dynamically.

ABOUT REGULAR POLYGONS

By David Dirkse



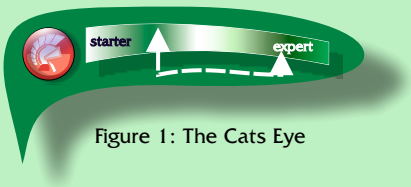
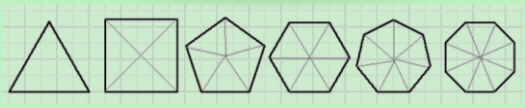


Figure 1: The Cats Eye

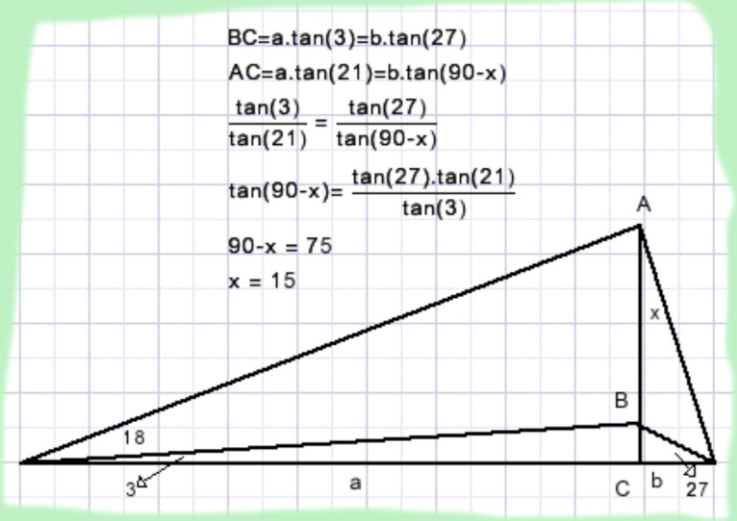
INTRODUCTION

In Euclidean geometry, a regular polygon is a polygon that is equiangular (all angles are equal in measure) and equilateral (all sides have the same length). Regular polygons may be either convex or star. In the limit, a sequence of regular polygons with an increasing number of sides approximates a circle, if the perimeter or area is fixed, or a regular apeirogon (effectively a straight line), if the edge length is fixed. Below are pictured some (3 to 8 edged) regular polygons.



Firgure 1

A regular N polygon may be considered as N identical isosceles triangles which top angle equals 360/N.
This Delphi project originated from a geometric problem.
Asked is to find the value of angle x in the picture below:



Firgure 2

The algebraic solution presented here needs a pocket calculator.
In plane geometry problems are solved by the application of theorems and analytic reasoning.
A geometrical solution however is difficult until we realize that all angles are multiples of three.
Angles of 3 degrees on a circle perimeter span arcs of 6 degrees of the circle according to the **theorem of Thales (Greek mathematician, 500BC)** Painting the triangle in the circumscribed circle of a 60 angled regular polygon shows the answer right away: (polygon edges not painted)

In geometry, **Thales' theorem** *states that if A, B, and C are distinct points on a circle where the line AC is a diameter, the angle ABC is a right angle.* **Thales's theorem** *is a special case of the inscribed angle theorem and is mentioned and proved as part of the 31st proposition in the third book of* **Euclid's Elements**. *It is generally attributed to* **Thales of Miletus,** *but it is sometimes attributed to* **Pythagoras**.
There is a very good example on the **Wiki** *site:*
https://en.wikipedia.org/wiki/Thales%27s_theorem#:~:text=In%20geometry%2C%20Thales'%20theorem%20states,book%20of%20Euclid's%20Elements.

WIKIPEDIA

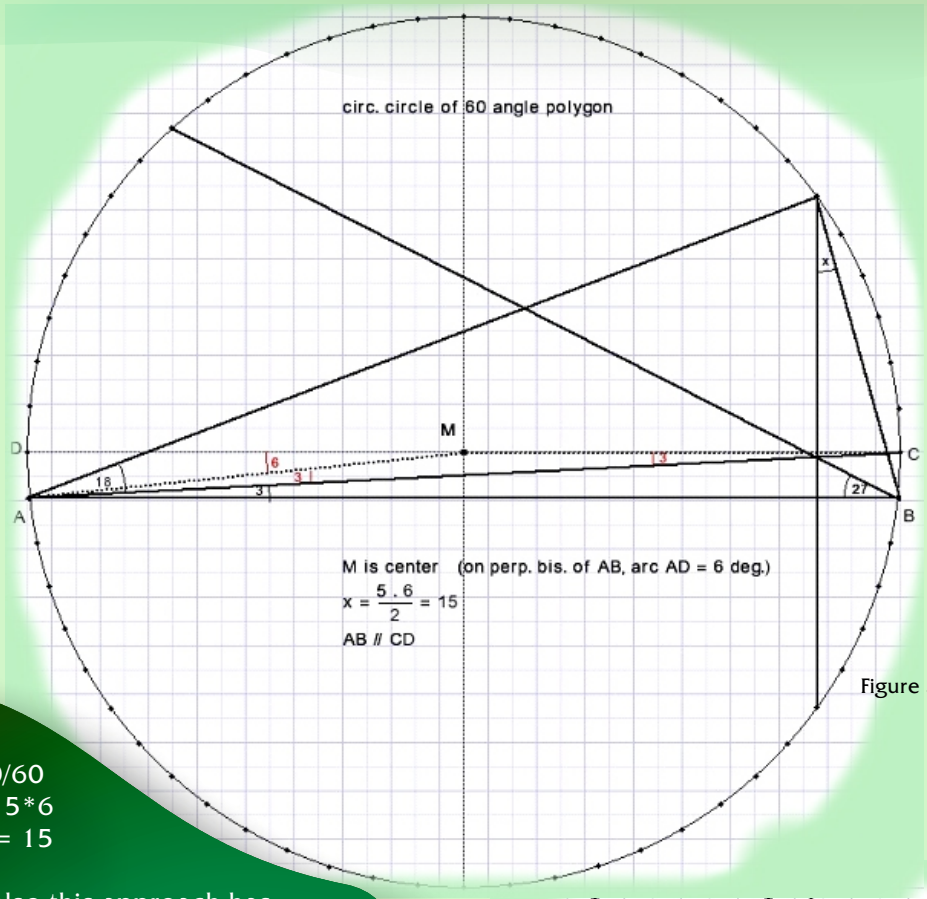


Figure 3

Each arc between points corresponds to 360/60 = 6 degrees. X spans 5*6 degrees so x = 30/2 = 15 degrees.
In many other cases also this approach has been useful. So I decided to build a project to investigate regular polygons.

INTERMEZZO 1
Angles may be measured as circle arcs. In the picture below, arc AB is expressed as angle at the circle center.
The theorem of **Thales states** that an angle (such as Firgure 4) on a circle

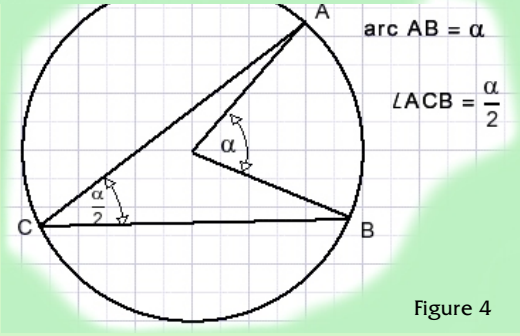


Figure 4

THE PROJECT

Below is a reduced picture of the project showing an 18 edged **polygon** with its diagonals and the circumscribed circle. Colored dots are painted at the intersection of diagonals. Their color indicate the number of diagonals crossing.

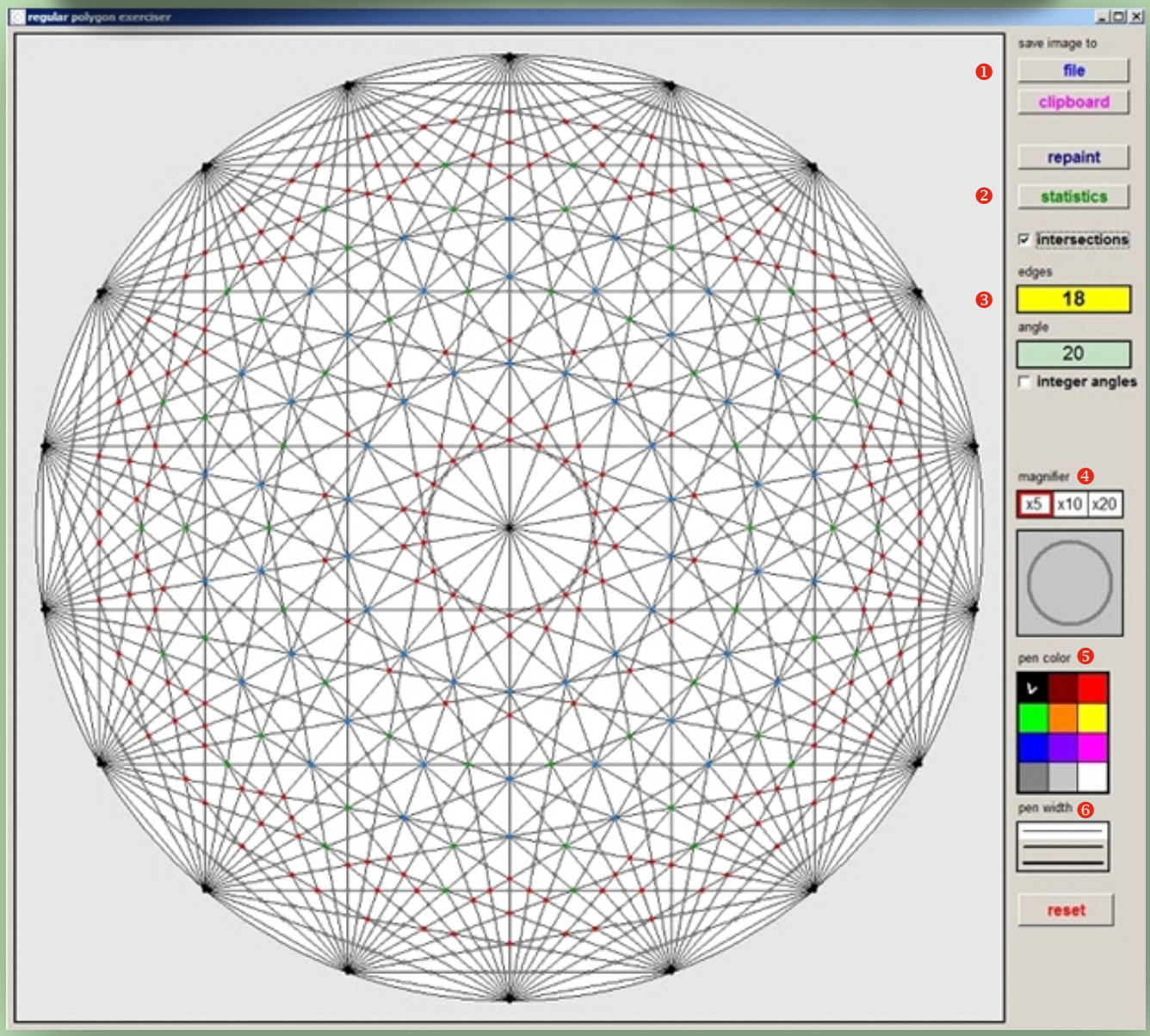


Figure 5

Buttons allow for

❶ Saving the image to a file or to the clipboard
❷ Showing statistics about intersecting diagonals
❸ Selecting the number of edges (3 to 60)
❹ Shifting a magnifying glass over the image to clearly observe the intersection of diagonals.
❺ Coloring an intersection according to the number of diagonals crossing
❻ Drawing lines with selected color and pen width

This article is about calculation and painting of the polygon, it's diagonals and intersections and also the operation of the magnifying glass. Added to that the readers high school math may be refreshed.

EDGE SELECTION
A TLabel component is used as a button. TLabel has OnEnter and OnLeave events, allowing background color change on action.

```
procedure TForm1.Label1MouseEnter(Sender: TObject);
begin
  label1.Color := $00c0ff;
end;

procedure TForm1.Label1MouseLeave(Sender: TObject);
begin
  label1.color := $00ffff;
end;
```

A left mouseDown event on the “edges” label increases the number of edges, a right mouseDown event decreases. To do the job a timer is started which runs as long as the button is pressed. This avoids clicking the button many times. Checkbox2.checked causes the edgecount to be limited to integer arcs.

```
function IncEdges : boolean; //increment edgecount
var E : byte;
begin
  if edgecount = maxEdge then result := false
  else
    begin
      E := edgecount+1;
      if form1.CheckBox2.checked then
        while (E < maxEdge) and (360 mod E > 0) do inc(E);
      setEdgecount(E);
      result := true;
    end;
end;

procedure setEdgeCount(newcount : byte);
const ff = '0.##';
begin
  edgecount := newCount;
  arc := 2*pi/edgecount;
  tanArc := tan(arc); //values needed later
  with form1 do
   begin
      label1.Caption := inttostr(edgecount);//show edgecount
      label3.Caption := formatfloat(ff,360/edgecount);//show arc
   end;
end;
```

INTERMEZZO 2

Traditionally, angles are measured in degrees where 360 degrees indicate a full circle turnaround. Reason for 360 is that this number has many divisors. The calculation of sine and cosine ratios is done by polynomials where the angle is expressed in radians.
360 degrees equals 2 x π radians, which is the perimeter of a circle with a radius of 1.
Note that the constant π always is an approximation. There does not exist a number or fraction which is exactly π.

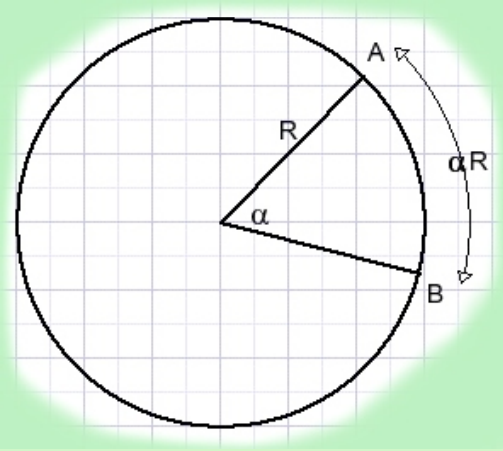


Figure 6

An angle of $\propto$ radians at the center of a circle with radius R spans an arc (AB) of length $\propto$ R.

CALCULATING THE POLYGON

Painting is done in a 1001 * 1001 pixels bitmap.
An odd number, so pixel[500,500] is the exact center.

The circumscribed circle has [500,500] as center, its radius is 480 pixels.

After painting, the bitmap is made visible by copying it to paintbox1 on form1.

A bitmap has coordinates [0,0] at the left top. In the case of painting we need the exact and absolute position of a pixel which is relative to the left top. These points however are rounded floating point values. Calculating intersections of diagonals requires precision, so floating point variables have to be used for the coordinates. Calculations become more simple regarding center pixel [500,500] as origin [0,0]. Here we use both methods.
The edges of the polygon are stored in the Alist array: { Alist[1] is the top, angles 2,3,.. run clockwise} The Alist array both holds integer and floating point values for the polynomial edge positions. The floating point values are relative to the center [500,500].

```
Const maxEdge = 60;
type TCpoint = record
        x,y : word;       //absolute pixel position
        rx,ry : single; //real value relative to screen center
       end;
var Alist : array[1..maxEdge] of TCpoint;//list of polygon angles

procedure Arc2XY(var x,y : single; a : byte);//a= 0,1,2
  //supply x,y coordinates of polygon angle
var na : single;
begin
  na := arc*(a-1);
  x  := 480*sin(na);
  y  := 480*cos(na);
end;

 Filling the Alist[ ] array:
 var i  : byte;
  x,y : word;
  rx,ry : single;
begin
   for i := 1 to edgecount do
     begin
       arc2XY(rx,ry,i);
          Alist[i].x := round(rx)+500;
          Alist[i].y := 500-round(ry);
          Alist[i].rx := rx;
          Alist[i].ry := ry;//UP +  DOWN – ; center relative
      end;
```

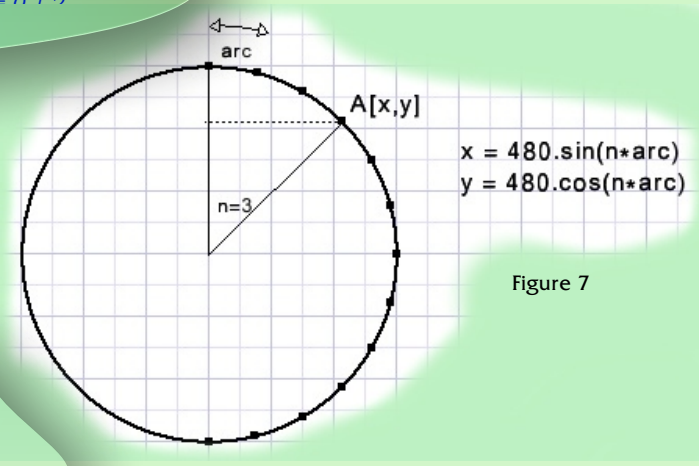


Figure 7

Please refer to the source code for painting of the edges and diagonals.
Two identical bitmaps are used:
Map1 holds the polygon, diagonals and circle.
Map2 is a copy of map1 and adds intersection points and also holds lines during drawing.

During the drawing process or while moving the magnifying glass, modified parts of **Map2** are erased by copying part of **Map1** to **Map2.** Modified parts of **Map2** are copied to `paintbox1` to become visible. This technique avoids erasing the paintbox which would cause flickering.

INTERMEZZO 3

The calculation of intersections is done with vector calculus.
Below is shown the vector equation of a line (AB):

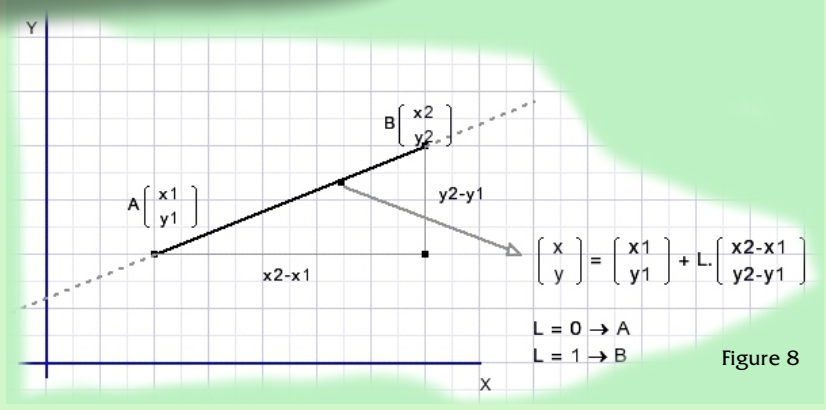


Figure 8

Variable naming (such as a1x):
```
//1 : begin of line, 2: end of line
//a : line through point A.    b : through point B
```
To find the intersection point of two lines:

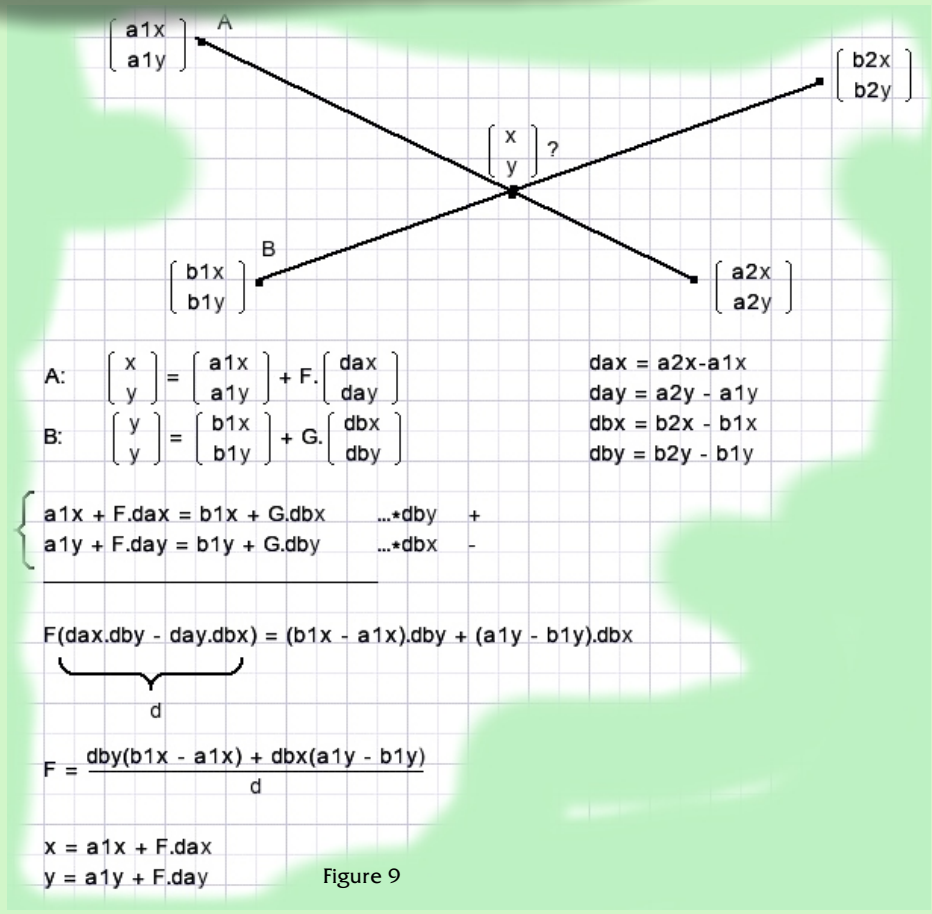


Figure 9

PROGRAM:
```
function GetIntersection(var x,y : single; a1,a2,b1,b2 : byte) : boolean;
// return intersection of diagonal a1–a2 and b1–b2 ;
   a1,2  b1,2 = 1,2,3…Alist index
// Return "false" in case of parallel lines (d = 0)
```

Saving time
An n- angled polygon has n(n-1)/2 lines (edges plus all diagonals).
A regular 60 angled polygon counts 1710 diagonals. To investigate intersections would require the examination of 1,461,195 line pairs. However, polygons have rotation symmetry.
All sections are the same. Only the diagonals that cross section 1 have to be examined.
Once knowing these intersection points the similar points in other sections may be calculated by rotation.

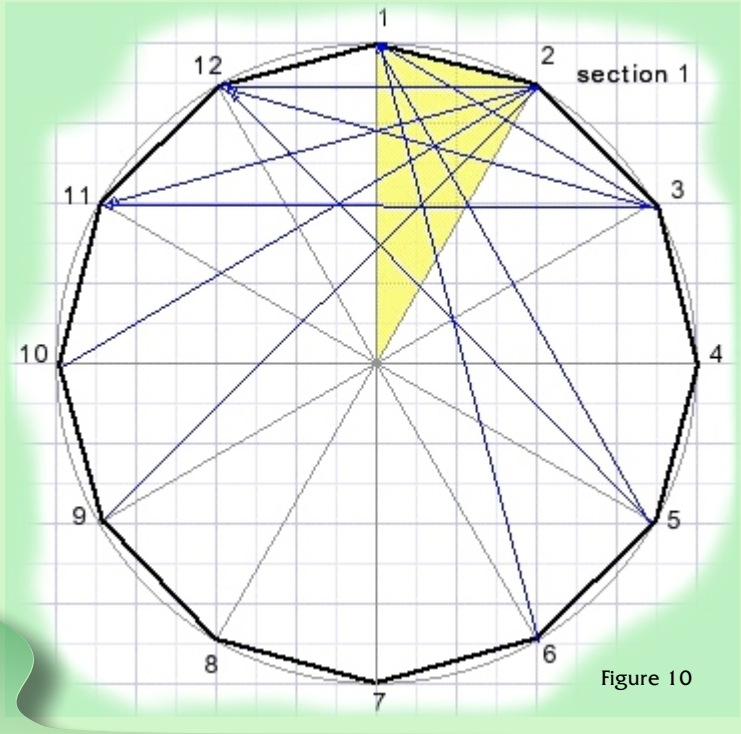


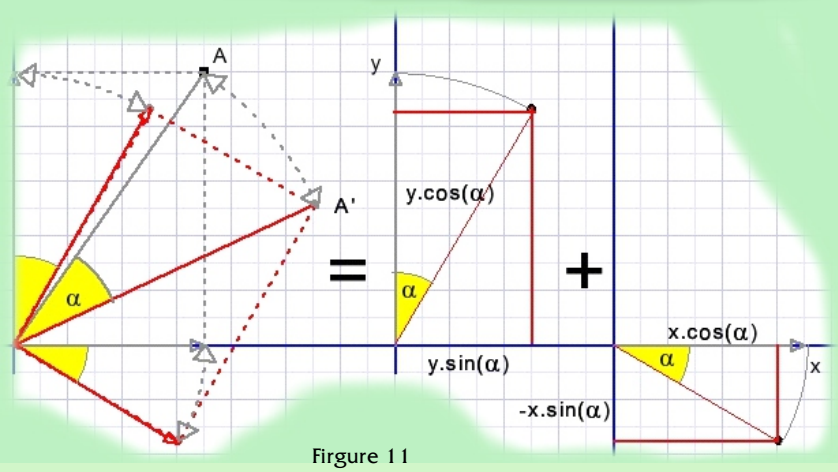
Figure 10

```
const INSTlistmax := 10000;
type  TIntersection = record
                        count : byte;
                      x,y : single;
                     end;
var  INTSlist : array[1..INTSlistmax] of TIntersection; //list of intersections in  section 1
```

Intersection points are added to above INTSlist. In case the (x,y) coordinates are already in the list , count is incremented.

INTERMEZZO(4)

Rotation of points.
A point A(x,y) is regarded the addition of its x and y coordinates.
X and Y are rotated separately, then the results are added. Rotation is clockwise.



Firgure 11

∝ becomes `x.cos(∝)`
and also decreases y by `x.sin(∝)`
Y becomes `y.cos(∝)`
and increases x by `y.sin(∝)`

Added together:
`A(x,y)` becomes `A'(x',y')`.
X becomes `x'= x.cos(∝) + y.sin(∝)`
Y becomes `y'= y.cos(∝) - x.sin(∝)`

Procedure :

```
function GetDotColor(c:byte):dword;
begin
 case c of
    2:result:= $808080;  //grey
    3:result:= $0000ff;  //red
    4:result:= $00b000;  //dark green
    5:result:= $ff8000;  //blue
    6:result:= $ffff00;  //light blue
    7:result:= $00c0ff;  //orange
    8:result:= $ff00ff;  //purple
    else result:= $00000000; //black
 end;//case
end;
```

```
procedure paintIntersections(sx,sy:single; count:byte);
  //paint intersection point (sx,sy) of chords in all segments
  //count : number of intersecting lines per point
var r:Trect;  x,y:word;  i:byte; a,sina,cosa,rx,ry:single;
  clr:dword;
begin
  clr:=getDotColor(count);
  with map2.Canvas do
  begin
    pen.Color:=clr;
    brush.color:=clr;
    brush.Style:=bsSolid;
    for i:=0 to edgecount-1 do
    begin
      a:=i*unit1.arc;//not arc procedure but variable
      sina:=sin(a);
      cosa:=cos(a);
      rx:=sx*cosa+sy*sina;
      ry:=sy*cosa-sx*sina;
      x:=round(rx)+500;
      y:=500-round(ry);
      r:=rect(x-2,y-2,x+3,y+3);
      ellipse(r);
      form1.PaintBox1.Canvas.CopyRect(r,map2.canvas,r);
    end; //with
  end; //for
end;
```

THE MAGNIFYING GLASS

The magnifying glass shows its portion of the screen 2, 5 or 10 times enlarged.

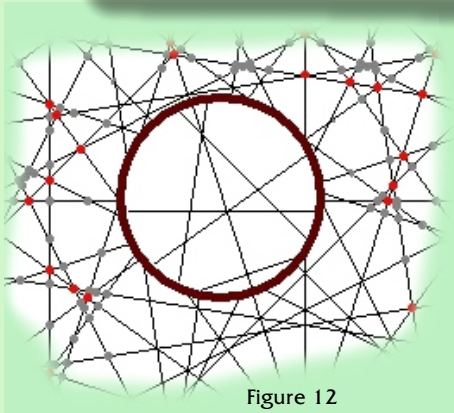


Figure 12

This is obtained by multiplying all coordinates by m (2,5,10) while calculating possible intersections with the magnifying glass circle. There is no actual enlargement of the picture at all, lines are recalculated. The magnifying glass radius is 55 pixels. To simplify calculations, all coordinates are shifted to make the magnifying glass center the origin [0,0] of the coordinate system. After calculations off course the coordinates are shifted back in place.
The picture on the next page shows the calculation of the intersection of a line and a circle.

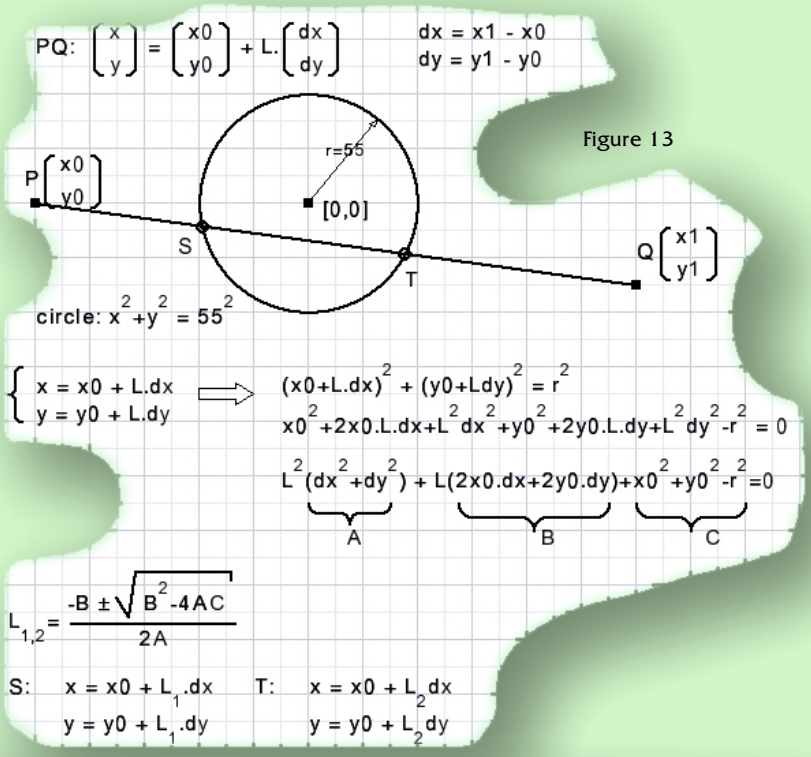


Figure 13

The result is line ST.
While moving, the magnifying glass center is [magX,magY] which are absolute pixel coordinates.

Before calculations:

```
Xoffset = (magX-500)*m
Yoffset =(magY-500)*m
X0 = m*Alist[i].rx - Xoffset
Y0 = m*Alist[i].ry - Yoffset
```

Now origin [0,0] is at the center of the magnifying glass.
For all diagonals a check is made for intersection with the glass.
If the root is negative in above calculations there is no intersection.
See `procedure paintmagnifierglass;` for details.

Keep in mind that the real `rx,ry` values in `Alist[ ]` are relative to `paintbox center [500,500]`.
Of course there is more to say. Such as the conditions for 3,4,…diagonals intersecting at one point, which must be based on symmetry.
But these considerations I save for other times. To use the magnifying glass, select the magnification, click on the glass which places it at the paintbox center. Shift the glass by placing the mousepointer over the glass, press mouse button and move mouse.

To remove the magnifying glass, click again on the button.
When the glass is not selected lines may be drawn by mouse movement. This may be useful in solving geometry puzzles which was the reason for this small project.

USING FLOATING POINT ARITHMETIC

Floating point values that are a power of 2 (such as 0.5 , 0.25) are exact values.
0.1 or π are approximations. Calculations using these values add inaccuracy.
In this project 32 bit "single" floating point variables are used. Their accuracy is 6 to 7 (decimal) digits.
Example:

```
Var a,b : single;
Begin
.....
 If a = b then .....//this will probably never be "true"
//  Instead this works
 If abs(a-b) < 1e-6 then .....// a almost equal to b
```

So at all times the programmer has to realize the amount of inaccuracy.

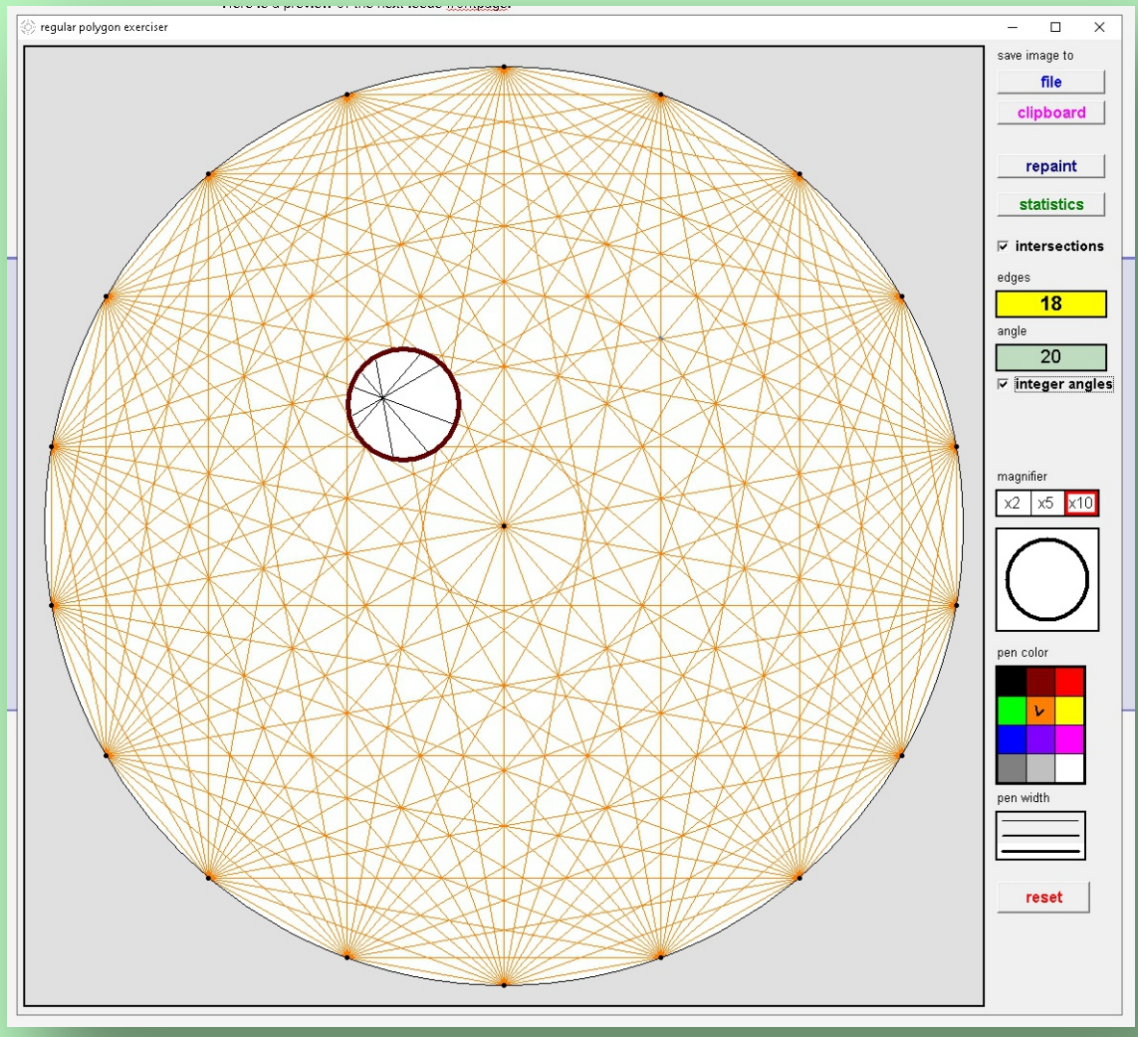
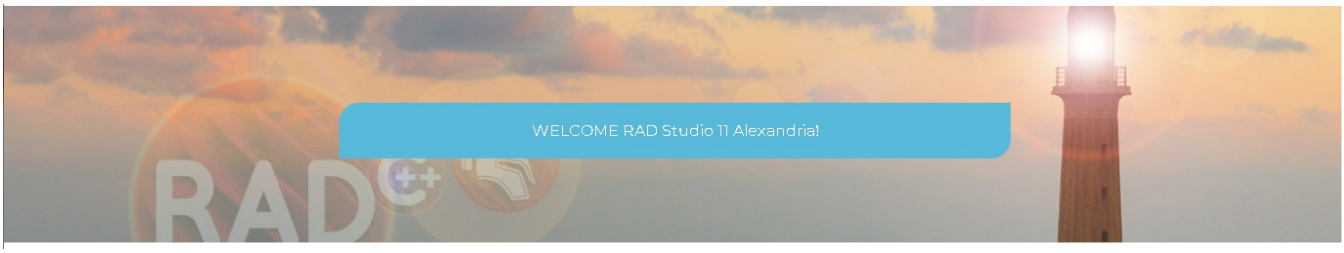


Figure 14

barnsten



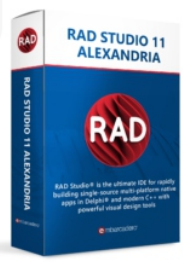
From
€1.529,00

Delphi 11 Alexandria

A powerful RAD environment for quickly developing high-performance native cross-platform applications using powerful visual design tools and integrated toolchains that independent developers and enterprise development teams love.

Shop Delphi

From
€2.699,00



RAD Studio 11 Alexandria

RAD Studio is the ultimate RAD environment loved by developers for quickly building high-performance native cross-platform applications in Modern C++ and Delphi using powerful visual design tools and integrated toolchains.

Shop RAD Studio

Phone number: +31 23 542 22 27

WINDOWS11 ON RASPBERRY P 4/8 GIG RUNNING DELPHI 11 AND LAZARUS 2.2.0RC1

ABSTRACT

Because my opinion is that we will go smaller and smaller with computers, having better CPU's and even more memory on board I wanted to show how far we have come already: It is now possible running **Delphi 11 on Windows 11** on **Raspberry. Lazarus** runs of course as well. If you want to try: take your time it will cost a few hours (4). But it works. For those who are interested we have a complete **ISO** prepared for you.

INTRODUCTION

In this article I try to explain how to install a **Raspberry Pi OS** for your **Raspberry Pi 4** card. It must be the 4 with 8 gig memory version because I want to install **Win 11** on it and then install Delphi and Lazarus. Do not try Windows 10!
This article is about **Windows 11**. The Raspberry PI is very hard to find so I'll give an address where you can order it.

`https://www.okdo.com/nl/p/okdo-raspberry-pi-4-8gb-basic-kit-universal-version/`

It is a trustworthy address from the UK. They only have the pack available: the **PI** itself is sold out for now, this kit contains an **SD card** which you will need to start with. For the windows version you will need a much faster card or rather a disk. I chose an **SSD** disk**:** they are fast booting and that' s what we want. 250 Gig should be working but you could try bigger.
Do not try this with an older version of the **Raspberry Pi** because there are chances you will raise errors because of time out.

Figure 1: The Raspi Kit in parts

To get started you will need some software which you can find at:
https://www.raspberrypi.com/software/
there is a video that might be helpful.
https://www.youtube.com/watch?v=ntaXWS8Lk34

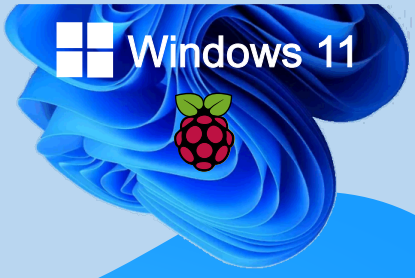
RASPBERRY PI OS

Your **Raspberry Pi** needs an operating system to work. This is it. **Raspberry Pi OS** (previously called **Raspbian**) is the official supported operating system.

Download and install **Raspberry Pi Imager** to a computer with an SD card reader. You can download the images for **Windows, Ubuntu** and **Mac** So what you need to install **Windows 11** on a **Raspberry Pi 4:**

- **Raspberry Pi 4 - 8 GB** memory on board – no less!
- 6GB or larger **microSD card** (available already in the kit)
- **Windows 11 PC**
- **USB** to **Ethernet** or **Wi-Fi dongle**
  Wi-Fi does not work with Windows on installing, even though there is 'WiFi' on board.
  Maybe we can find a way later to handle this , but for now you you would need a **WiFi dongle**.
- **Bluetooth** is available
- Keyboard, mouse, **HDMI** cable (available already in the kit) and power supply 3Volt (available already in the kit) for your **Raspberry Pi**.

Figure 2: The Raspi Kit is very easy to build, and later on quite helpfull



WIKIPEDIA

EXPLANATION OF THE ARM WORKINGS FOR PROGRAMS.

(ARM(previously an acronym for Advanced RISC Machines and originally Acorn RISC Machine) is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments. Arm Ltd. develops the architecture and licenses it to other companies, who design their own products that implement one of those architectures—including systems-on-chips (SoC) and systems-on-modules (SoM) that incorporate different components such as memory, interfaces, and radios. It also designs cores that implement this instruction set and licenses these designs to a number of companies that incorporate those core designs into their own products.)

WINDOWS 11

is compatible with most ARM devices made today except Snapdragon 835 devices.

Through Windows 11, Microsoft has made it easier for developers to create apps that run natively on ARM.
The main problem with Windows 10 devices equipped with ARM processors is the lack of apps. This is because these devices only support 32-bit emulation.
That's actually a significant limitation for many users.

Now Windows 11 brings support for 64-bit apps as well!
Windows 10 on ARM uses a special ARM64 system called **CHPE,** acronym for **"Compiled Hybrid Portable Executable"**.
CHPE is rather complex and thus not easy to understand and use.
Windows 11 makes x64 Emulation on **ARM** possible.
Windows 11 replaces **CHPE** with **ARM64EC (Emulation Compatible).**
Thanks to this new application binary interface, all plug-ins are compatible with the ARM64EC code. It doesn't matter if they're ported to ARM64 or not. For more information, see Using ARM64EC to build apps for Windows 11 on ARM devices:

https://docs.microsoft.com/en-us/windows/uwp/porting/arm64ec

This means that programs that need third-party plug-ins, can be ported to Windows on ARM without any problem. Developers do not need to remove extra plug-ins when porting their apps.
ARM32 apps run just fine on Windows 11.
Windows 11 is compatible with the majority of the ARM-based devices. The OS relies on a new application binary interface called ARM64EC (Emulation Compatible), making it easier to develop apps that run natively on ARM.
Microsoft signalled the importance of compatibility for Windows on Arm and turned on 64-bit emulation in Insider builds. With Windows 11, it is possible to create run and test through developers their already designed apps. Because the system recognizes arm or emulates "normal" windows apps it is a platform which can be easily deployed from now on. I will test the system by creating native ARM and run them on the same platform: Windows11 on Raspberry Pi.

So the plan to
install all this is:

1. Set up the Raspberry Pi OS – the first time on Mini SD Card.
   We need that to be able to make installer (BootSequence) arrangements for the Windows 11 OS
2. Create the installer for windows.
3. Unzip the windows environment to a disk, so you can start to install it.
4. Create your very own Windows environment. All you nee is to have either already a version of Windows 10 or 11and have a Microsoft account.

Once all is done you can start on loading – (installing) your windows programs that you want in our case Delphi and Lazarus. We will start with Lazarus because that is a quick install.

❶
RASPBERRY PI OS

Your **Raspberry Pi** needs an operating system to work. That is it. **Raspberry Pi OS** (previously called Raspbian) is the official supported operating system.

`https://www.raspberrypi.com/software/`

INSTALLING
RASPBERRY PI OS
USING RASPBERRY PI IMAGER

Raspberry Pi Imager is the quick and easy way to install **Raspberry Pi OS** and other operating systems to a **microSD** card, ready to use with your Raspberry Pi.
Watch the45-second video to learn how to install an operating system using Raspberry Pi Imager.

`https://www.youtube.com/watch?v=ntaXWS8Lk34`

Download and install the Raspberry Pi Imager to a computer with an **SD Card Reader**. Put the SD card you'll use with your Raspberry Pi into the reader and run **Raspberry Pi Imager.**
The package containes already an SD Card:

Figure 3: The SD Card Reader and the included SD Card

There are three downloads on this page:
`https://www.raspberrypi.com/software/`
Download for macOS
Download for Windows
Download for Ubuntu for x86

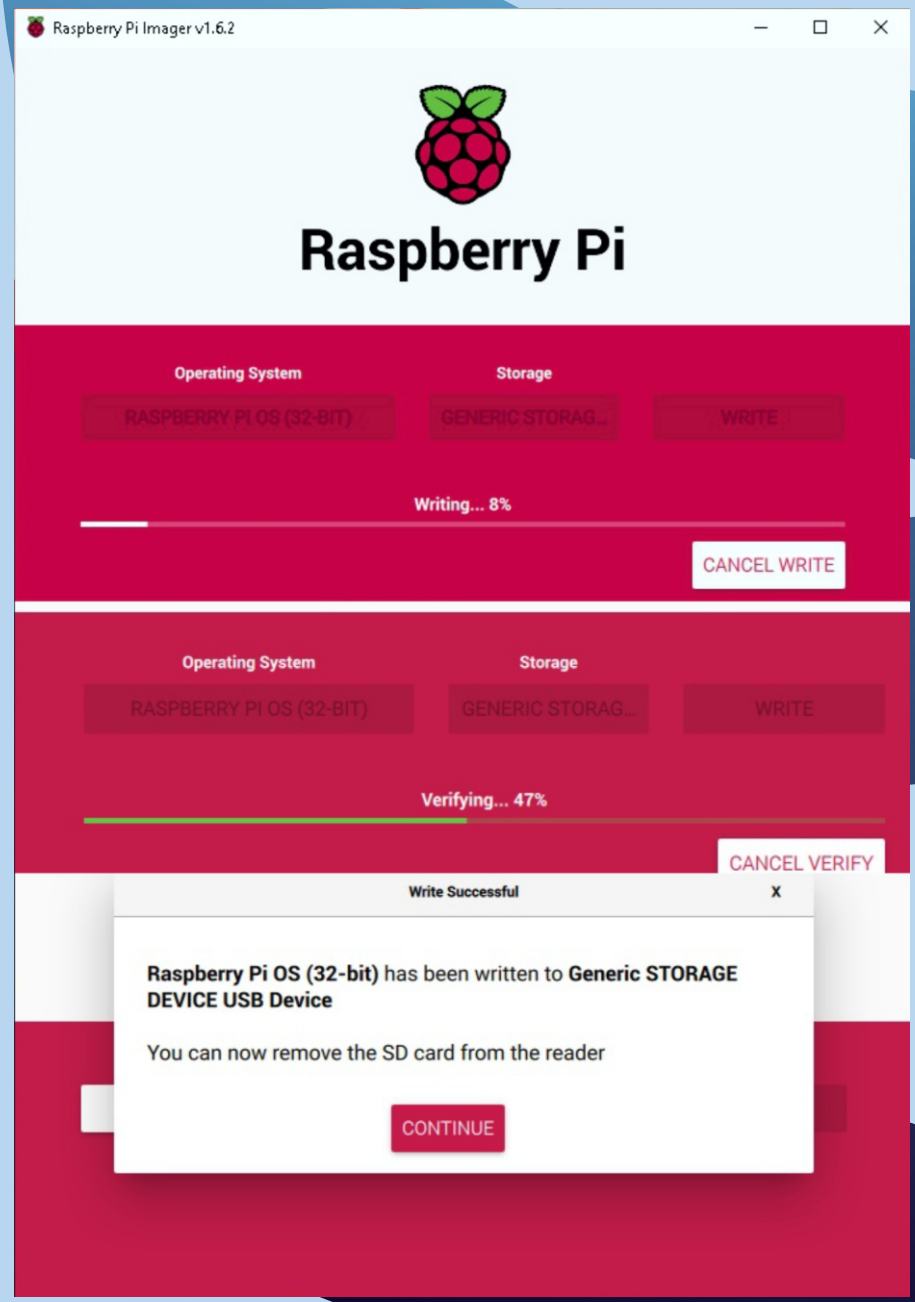


Figure 4: The installation starts

Figure 5: The OS has been written to the DS card. The RPi Imager has a hidden advanced options screen to set WiFi, press CTRL + SHIFT + X
https://www.tomshardware.com/news/raspberry-pi-imager-now-comes-with-advanced-options

Because the OS is not yet available I made some screenshots with a camera...

Figure 6: The os starts up and shows the necessary steps to make selections

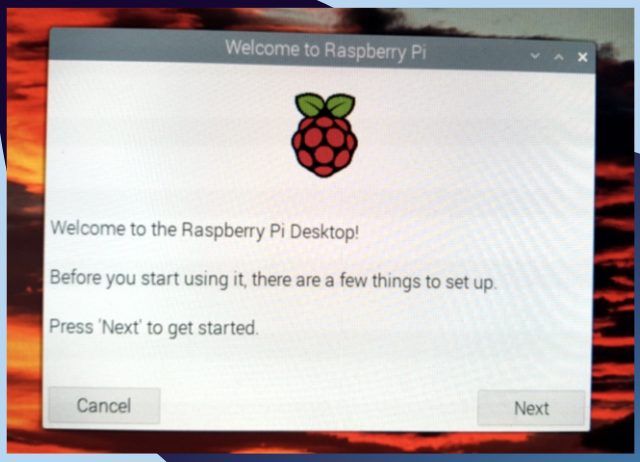


Figure 7: Create a password: don't forget to write it down for later

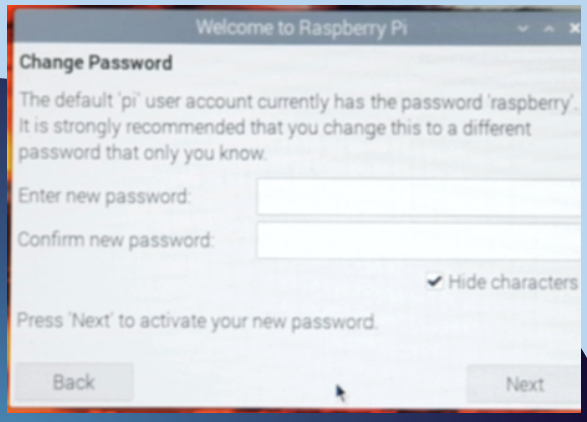


Figure 8: Localisation

Figure 9: Set up screen

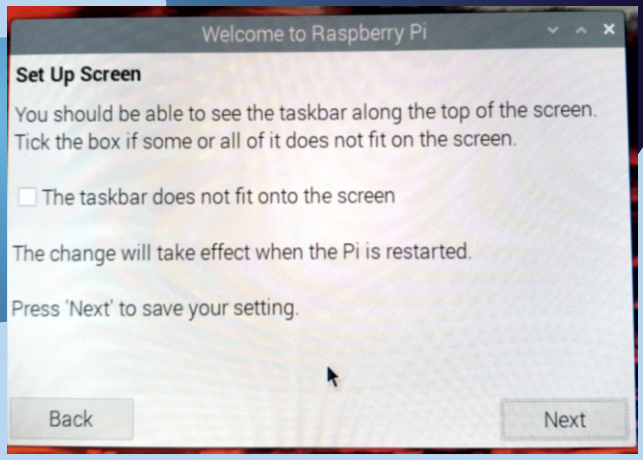


Figure 10: The wifi works, but later under WIN11 it does NOT

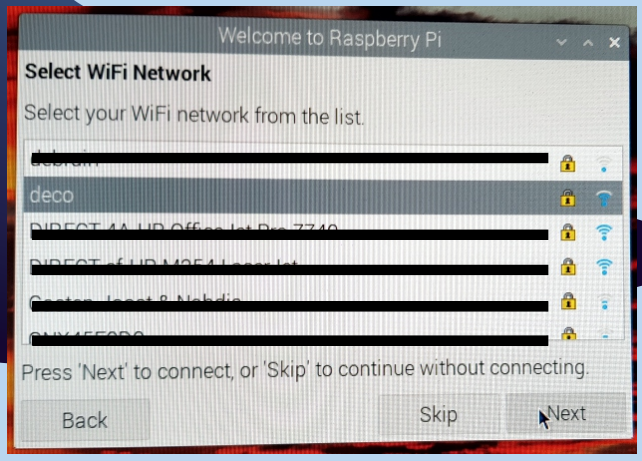


Figure 11: press restart

Figure 12: Updating to the latest version

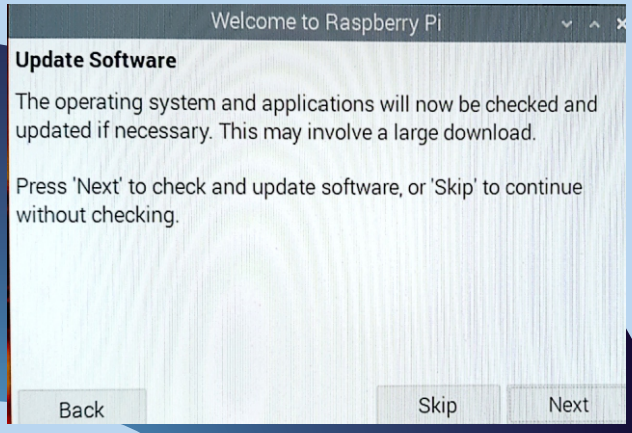
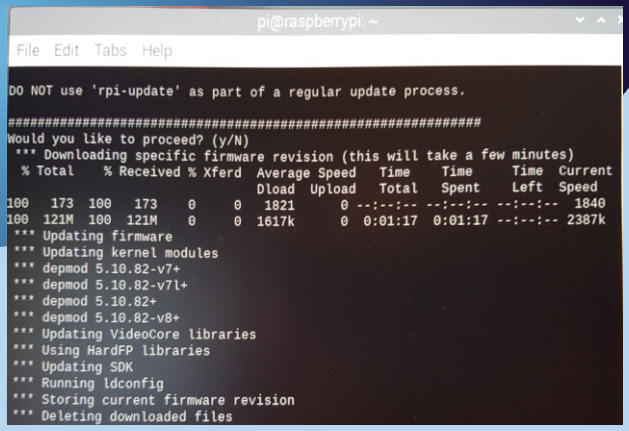


Figure 13: Running the updates



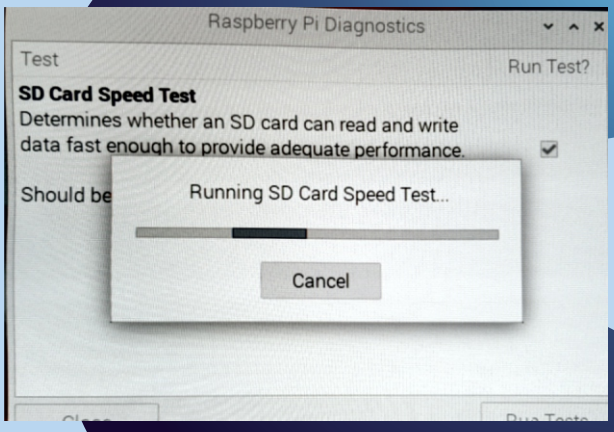


Figure 14: You can try the speed test but it is not necessary

The

The **Raspberry**
is no **Microsoft** product and thus there are no guarantees for the security and stability of the installation. The OS does work surprisingly well, but is missing a number of important parts.
The built-in **Wi-Fi, Bluetooth** and **GPIO** * connectivity do not yet work under **Windows,** so you best use Ethernet or a USB dongle for **Wi-Fi** should be used for an internet connection.

WikipediA

*(*A general-purpose input/output (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by the user at runtime.
GPIOs have no predefined purpose and are unused by default. If used, the purpose and behaviour of a GPIO is defined and implemented by the designer of higher assembly-level circuitry: the circuit board designer in the case of integrated circuit GPIOs, or system integrator in the case of board-level GPIOs.)*

Aud

HDMI (audio)
is also not available, but it does work via the 3.5mm jack.

WikipediA

(High-Definition Multimedia Interface (HDMI) is a proprietary audio/video interface for transmitting uncompressed video data and compressed or uncompressed digital audio data from an HDMI-compliant source device, such as a display controller, to a compatible computer monitor, video projector, digital television, or digital audio device. HDMI is a digital replacement for analog video standards.)

PREPARATION

To start with, you must check whether the latest firmware and bootloader of the Raspberry Pi are installed in order to boot from USB. This can only be updated via the official OS, not Windows. If you choose to use an SD card instead of an SSD, you can skip this section.

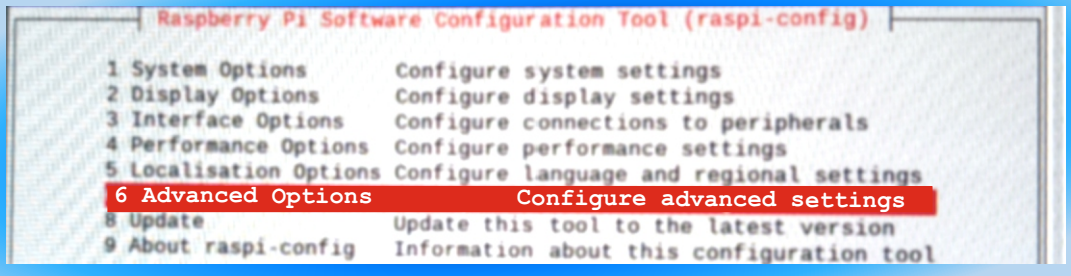
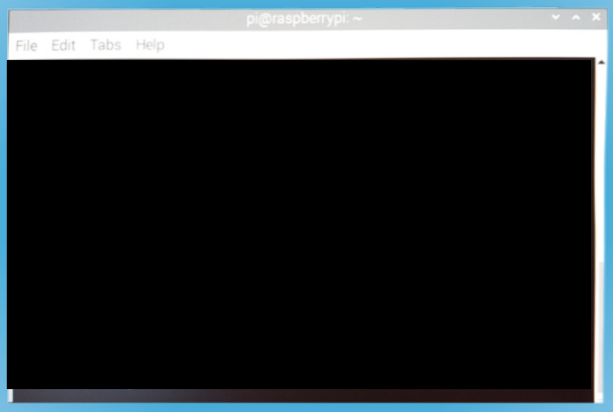


Figure 15: 6 Advanced options Configure advanced settings

Boot the Pi 4 from an SD card with Raspberry Pi OS and open a terminal. Enter the following to update the operating system and firmware:

```
sudo apt update
sudo apt full-upgrade
sudo rpi-update
```

Figure 16: The terminal window



❶ Reboot the Pi and install the latest bootloader as follows:

```
sudo rpi-eeprom-update -d -a
```

❷ Reboot again and open raspi-config:

```
sudo raspi-config
```

❸ Select Boot Order, press Enter, choose network or USB device boot and press enter

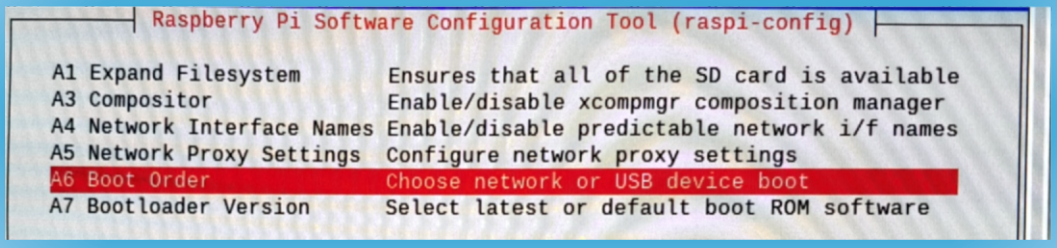


Figure 17: Boot Order - USB

❹ Now select USB Boot, boot from usb if available.Otherwise boot from SD crd .

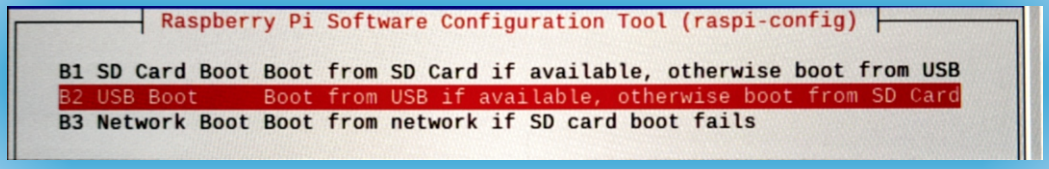


Figure 18: Boot from USB, where your drive is connected to (SSD)

6. Select **Finish,** then **No** when prompted to reboot.

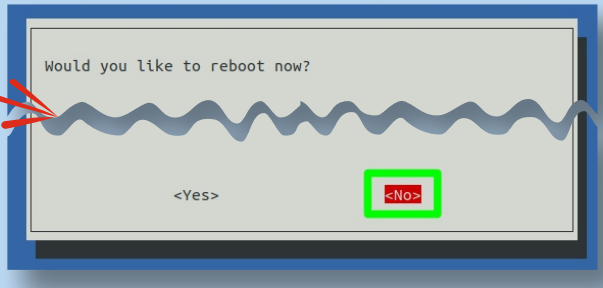


Figure 19: Do Not Reboot! (yet)

Go to this site
`https://uupdump.net/`
see details on page 11/12/13/14

❶ **Search UUP dump for "Windows 11 arm"**
and choose the latest version. (See page 11 of this article)

❷ **Select the desired language** and choose the edition you want to install,
in this case Pro has been chosen. (See page Figure 12 top)

❸ **Set the download method** as "Download and convert to ISO",
then click "Create download package".(See page Figure 13)

❹ **Extract the download** to a new folder named `'Win11'`
and navigate there. You will probably need to create this directory.

❺ **Double click on the** `uup_download_windows.cmd` file, (See page Figure 13)
this will display a security warning.
Choose **More Info/More information,** then **Run Anyway/Run Anyway.**
Finally, allow the app to make changes to your device.

❻ **Now a command prompt will open showing the output of a program.**
This downloads and patches the **Windows 11** image and then prepares the iso.
When the process is complete, you can press 0 to close the prompt.
(See page Figure 14last line)

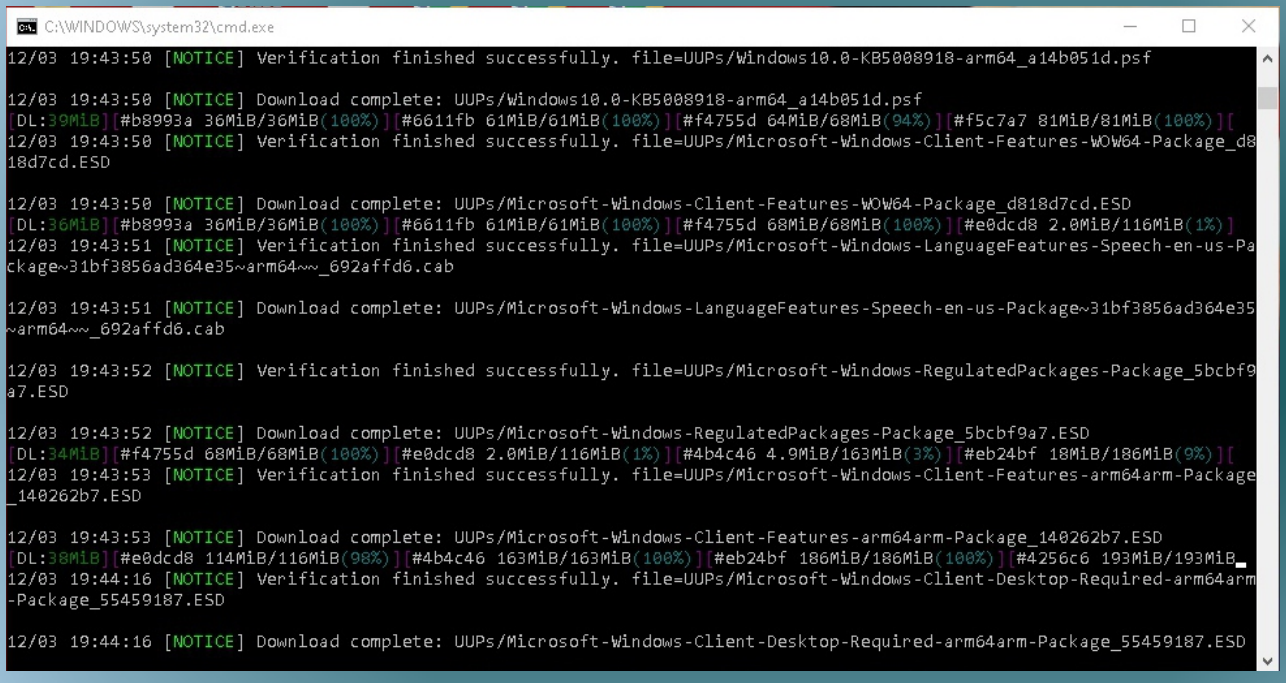


Figure 20: Running the download of the Microsoft packages.

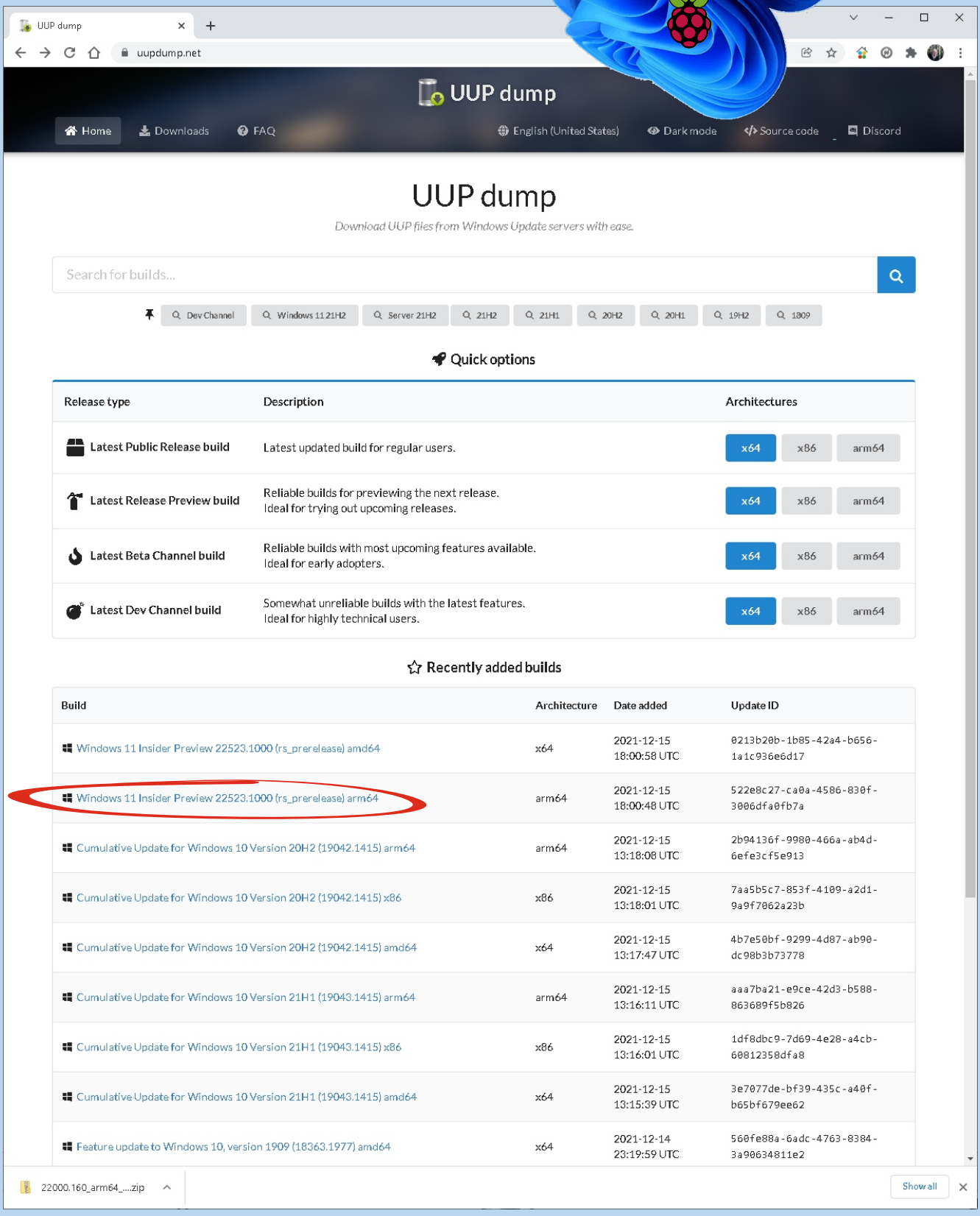


Figure 20: The choice to make: windows Insider Preview xxxxx (rs_prerelease) arm 64

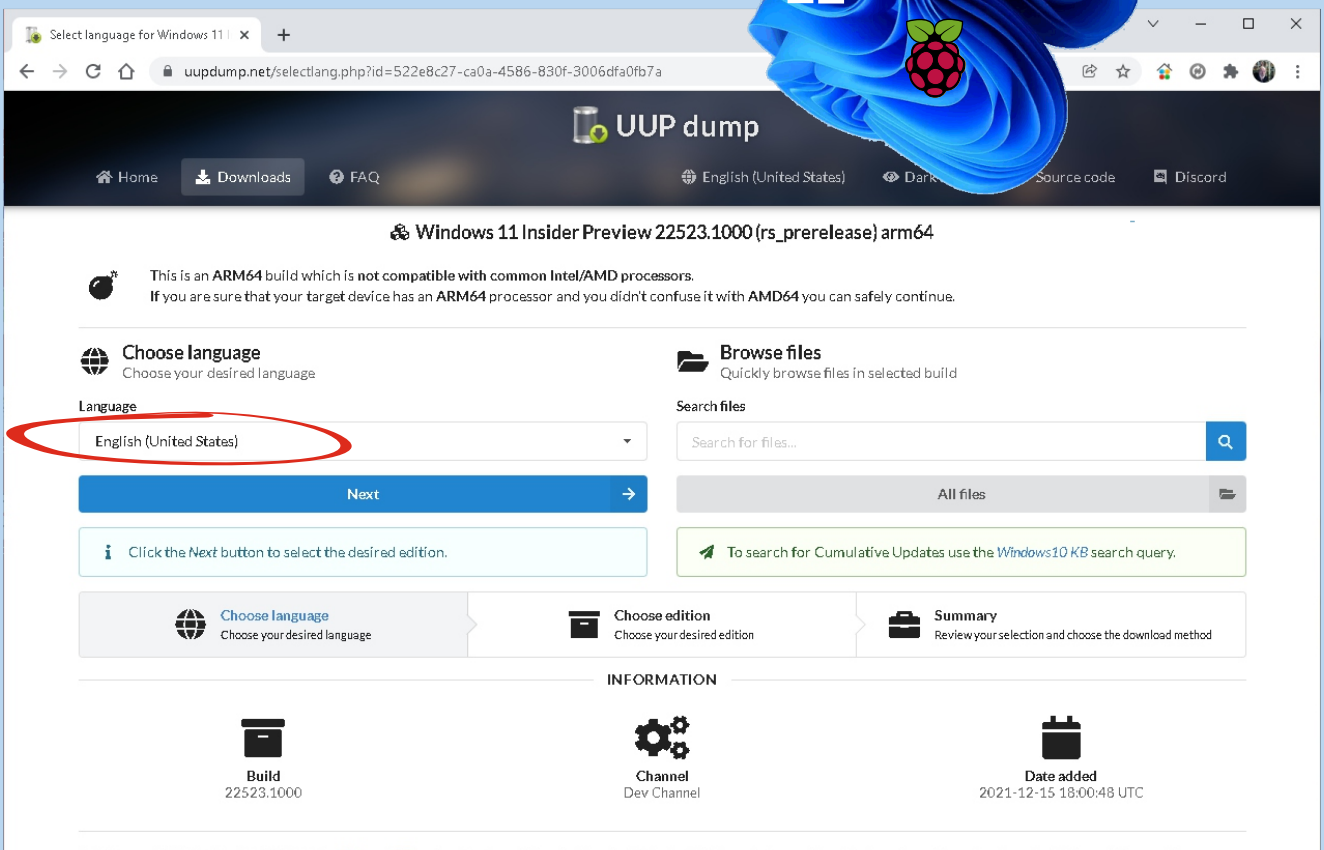


Figure 21: Choose the language you prefere.

Figure 22: Choose your Edition

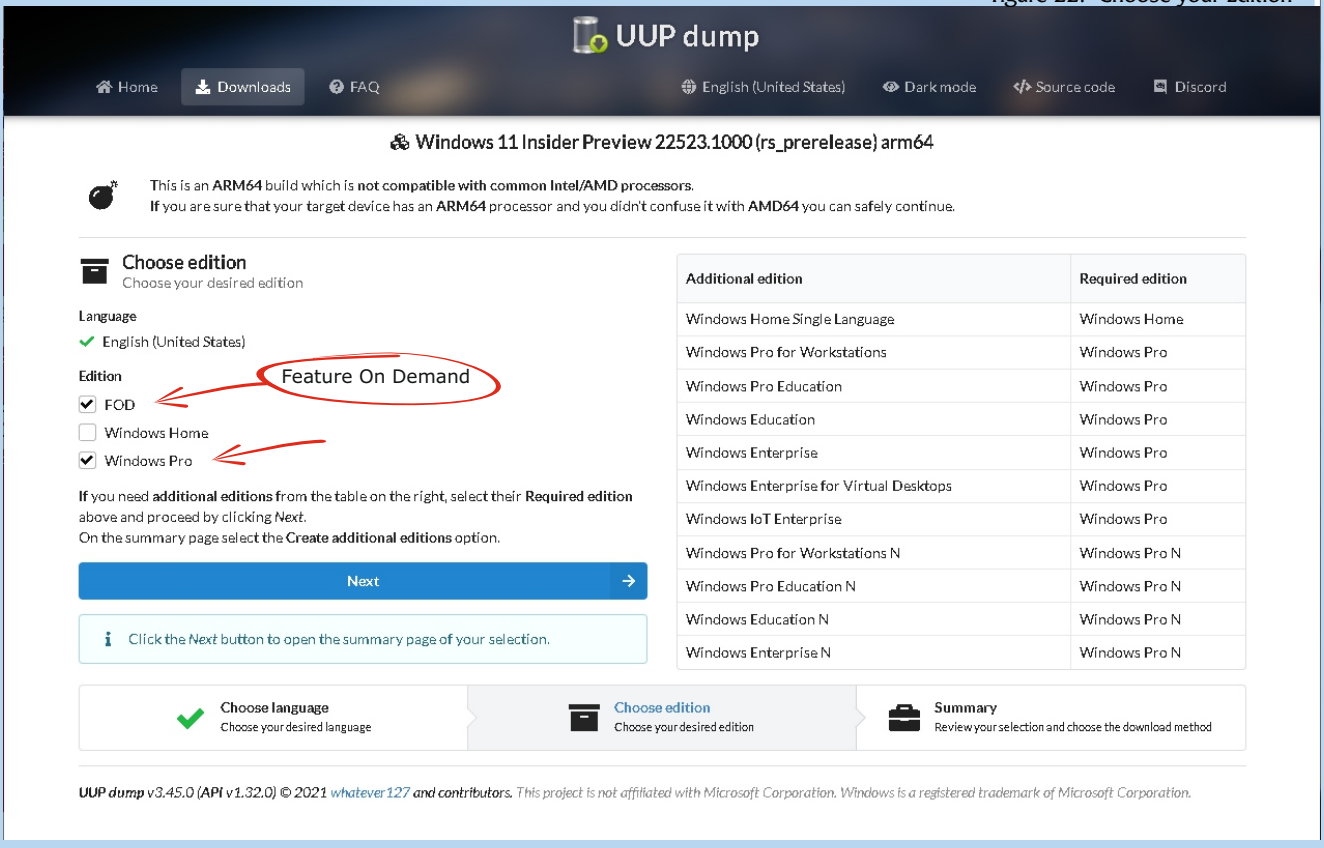
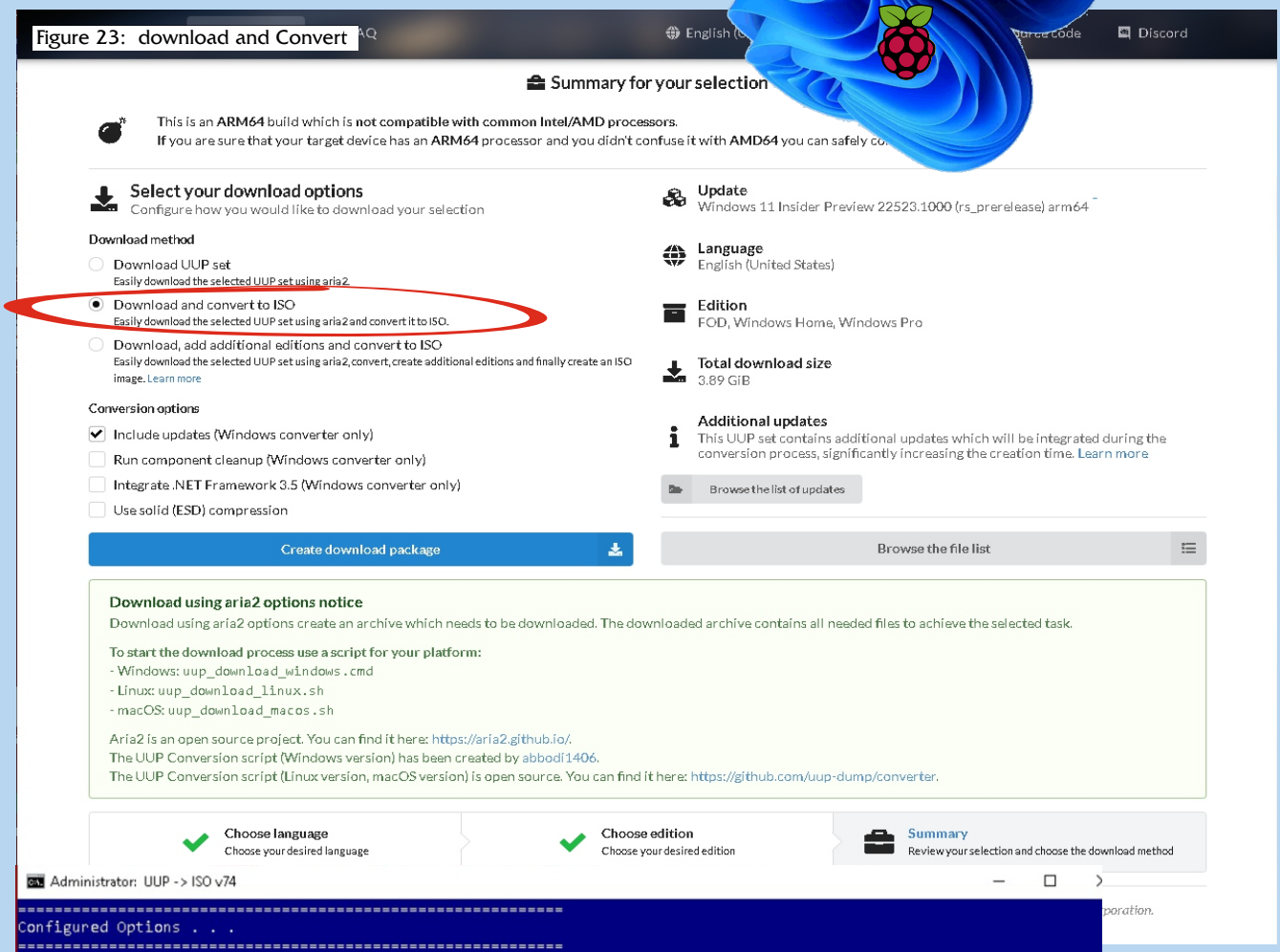


Figure 23: download and Convert



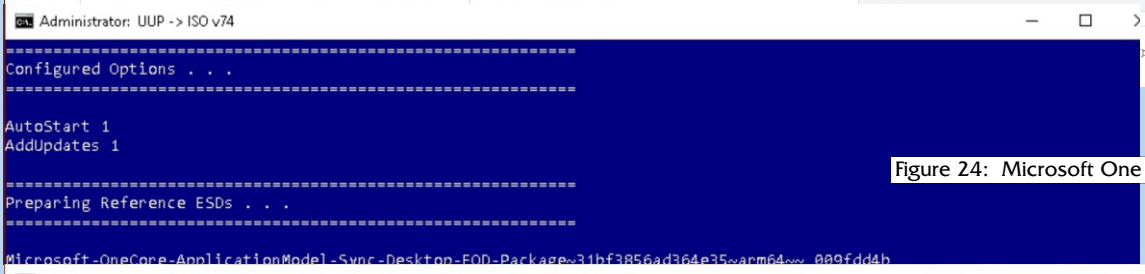


Figure 24: Microsoft One core

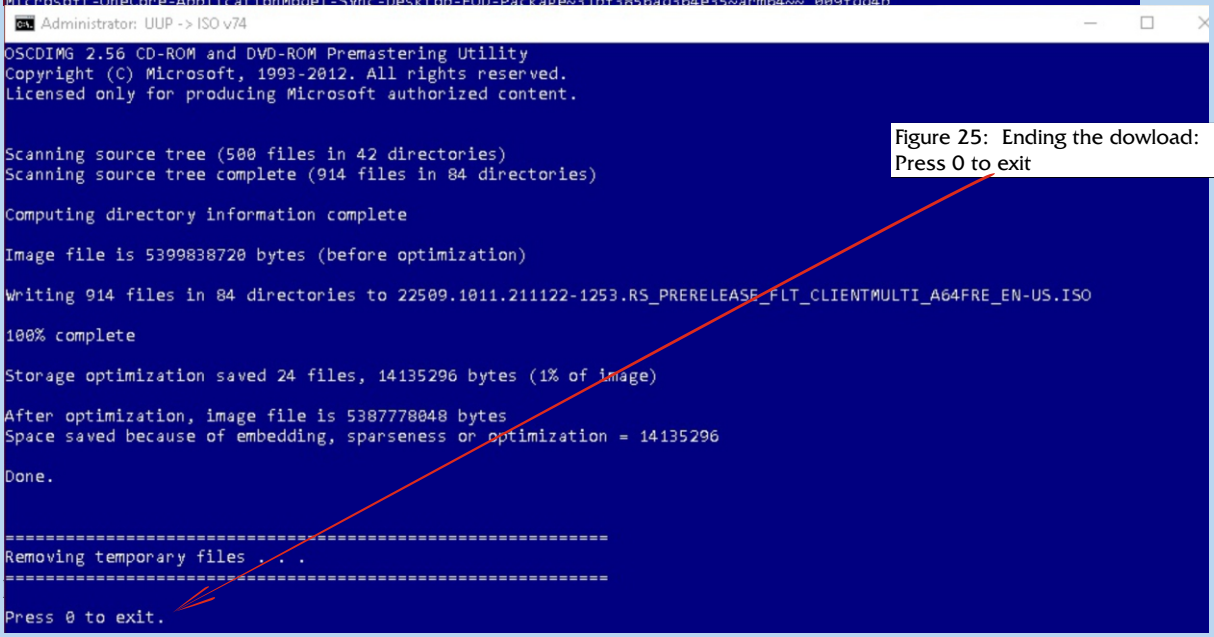
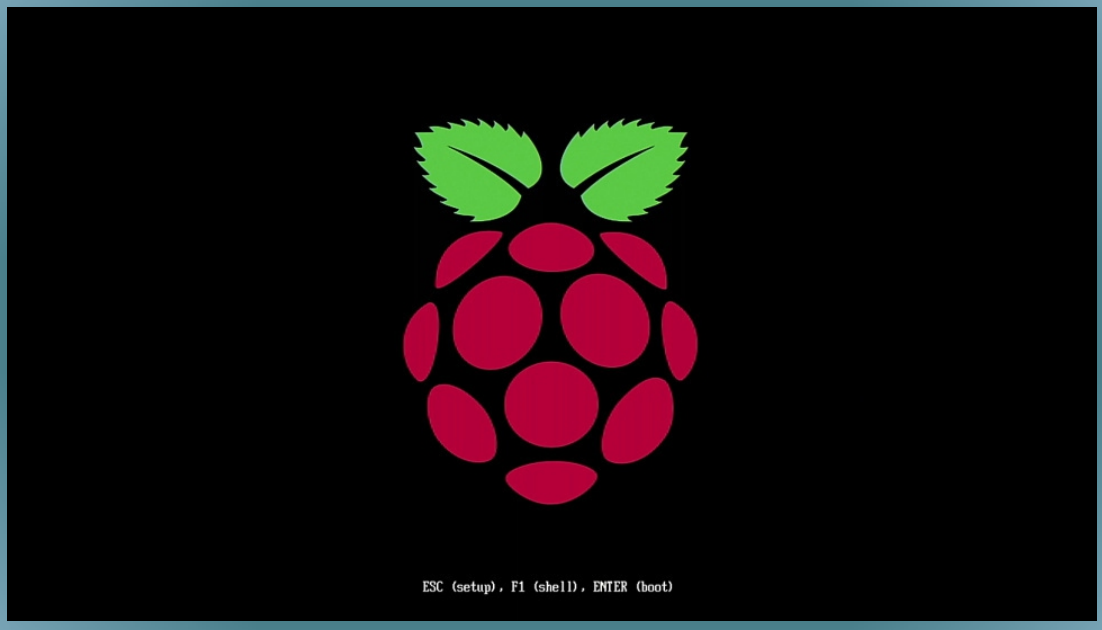


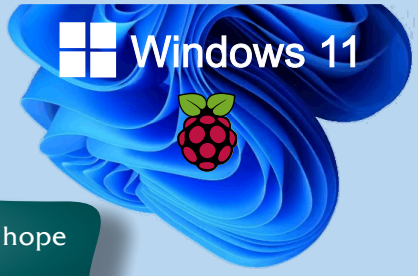
Figure 25: Ending the dowload: Press 0 to exit

Now you have an ISO image that you can put on an SD card or USB disk.
(For extra information see next page).

This works as follows:
Put your storage medium of choice into the computer and check that it is ready for use.
Please note, the drive will be formatted and all data on it will be deleted.

1. **Download the Windows on Raspberry imager** and extract the zip file to the previously created win11 folder.
2. Open **WoR** (Windows on Raspberry) and allow the application to make changes to your system. Choose a language and press Next.
3. Select the drive you want to use and the device type, in this case a Raspberry Pi 4/8 gb mem. Press Next to continue.
4. Select the new **Windows 11 iso** and continue. Choose the latest drivers and firmware available on the server, these will be stored locally. Continue to use the chosen configuration.
5. In the installation overview, verify that the correct disk and device type are selected. Click Install to start the process. This takes about 10 minutes with a **USB SSD,** for a **microSD** card it takes a bit more time.
6. When the installation is complete, **WoR** can be closed. Eject the drive and connect it to your Raspberry Pi. Also connect your peripherals and boot the Pi.





It might be interesting for you to look at some extra information:
Here are the URLs to get some more information, but I hope this all will not confuse you.

```
https://www.microsoft.com/en-us/software-download/windowsinsiderpreviewarm64

https://www.tomshardware.com/how-to/install-windows-10-raspberry-pi

https://www.techrepublic.com/article/
what-windows-11-means-for-windows-on-arm-and-why-it-will-bring-more-big-name-apps/
```

WIKIPEDIA

(*A Uniform Resource Locator (URL), colloquially termed a web address, is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it. A URL is a specific type of Uniform Resource Identifier (URI), although many people use the two terms interchangeably. URLs occur most commonly to reference web pages (http) but are also used for file transfer (ftp), email (mailto), database access (JDBC), and many other applications*)

For the USB connection it is advisable to use a special cable that has a USB connector and at the other end a SSD Data connector.
Difficult to find? Probably you have the solution already at home:
if you have a USB Backup system (see Figure 26) or what I had was an enclosure external Drive connection (see Figure 27) where you could mount your SSD.

Figure 26: Ewent dockingstation

Figure 27: Sweex USB enclosure for 2.5" SATA HDD

```
https://www.ewent.com/en-us/products/usb-hard-drive-enclosures/

https://www.ewent.com/en-us/usb-3-0-hdd-dual-docking-station-
ew7014?returnurl=%2fen-us%2fproducts%2fusb-hard-drive-enclosures%2f%3fcount%3d20
```

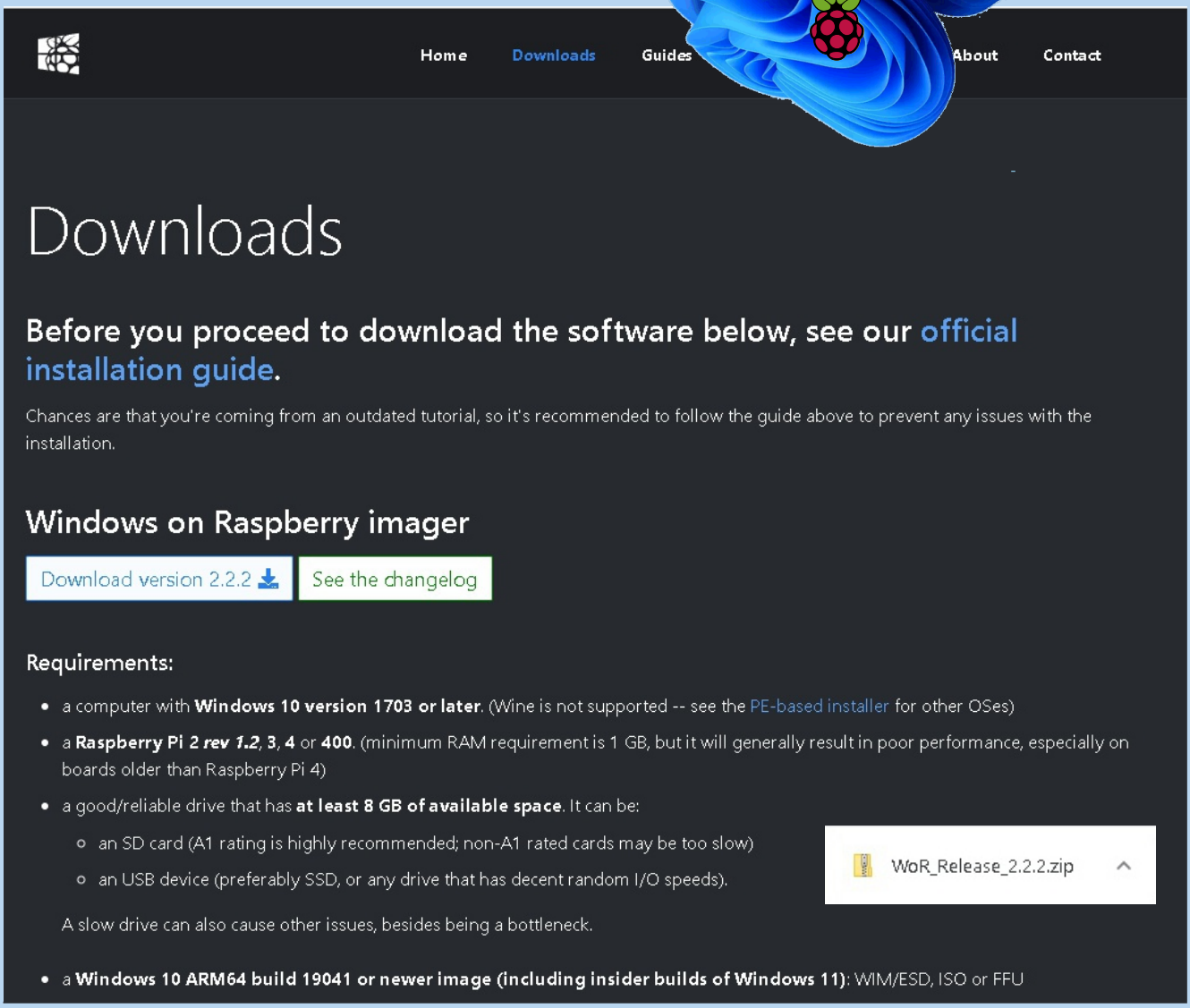
With the drive connected and the **Raspberry Pi 4** booted up, you should now go through the standard setup sequence.
After the installation process and a few tweaks, your Pi is then ready to use with **Windows 11.**

1. Boot the **Raspberry Pi** and press **ESC** when prompted.
2. Scroll to **Device Manager** and press Enter, then go to **Raspberry Pi Configuration**, then to →**Advanced Configuration.**
3. **Set "Limit RAM to 3 GB" to "disabled"** and press **F10** to save, go back with Escape.
4. Open Display Configuration and set the correct resolution by scrolling to it and pressing Enter. Use **F10** again **to save and press Escape.**
5. In **CPU configuration the processor can be overclocked,** make sure the frequency is now **set to Default** otherwise **Windows 11 cannot boot!**
6. Press **ESC** until you are back in the first menu, then choose **CONTINUE** to exit the **UEFI** and **REBOOT.**

Figure 28: Installing Win11

WIKIPEDIA

The **Unified Extensible Firmware Interface (UEFI)** *is a publicly available specification that defines a software interface between an operating system and platform firmware.* **UEFI** *replaces the legacy* **Basic Input/Output System (BIOS)** *firmware interface originally present in all IBM PC-compatible personal computers, with most* **UEFI** *firmware implementations providing support for legacy BIOS services.* **UEFI** *can support remote diagnostics and repair of computers, even with no operating system installed.*



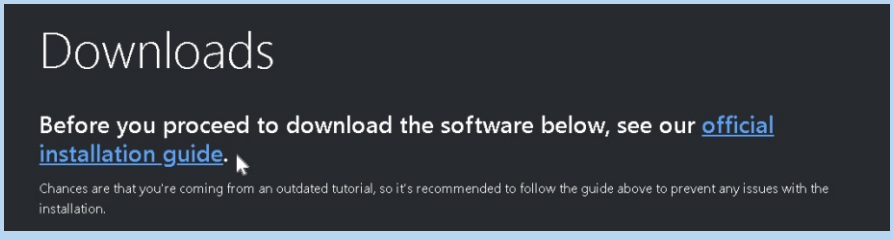


Figure at the top 30:
Downloading Windows 11
Windows On Raspberry imager (WOR)

Figure at the left 31:
Installation Guide

Figure at left bottom 32:
Choose your operation system

Now follow the instructions

Figure 33:

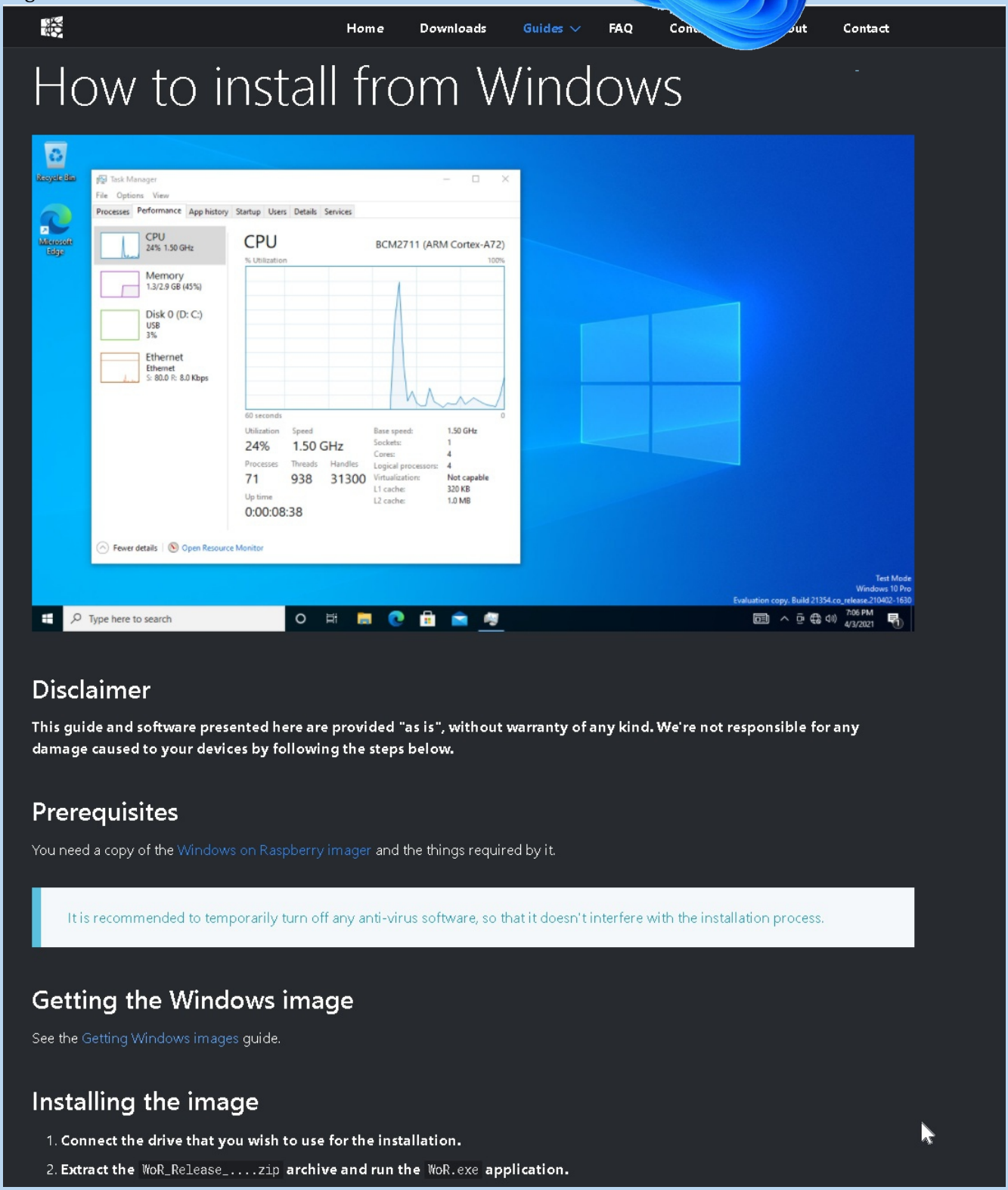


Figure 34:

Figure 35:

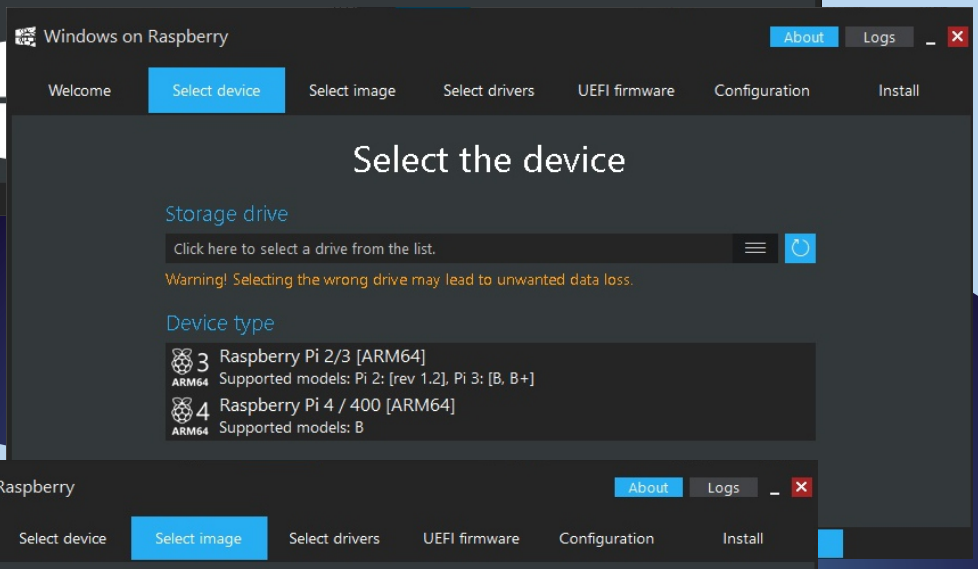


Figure 36:

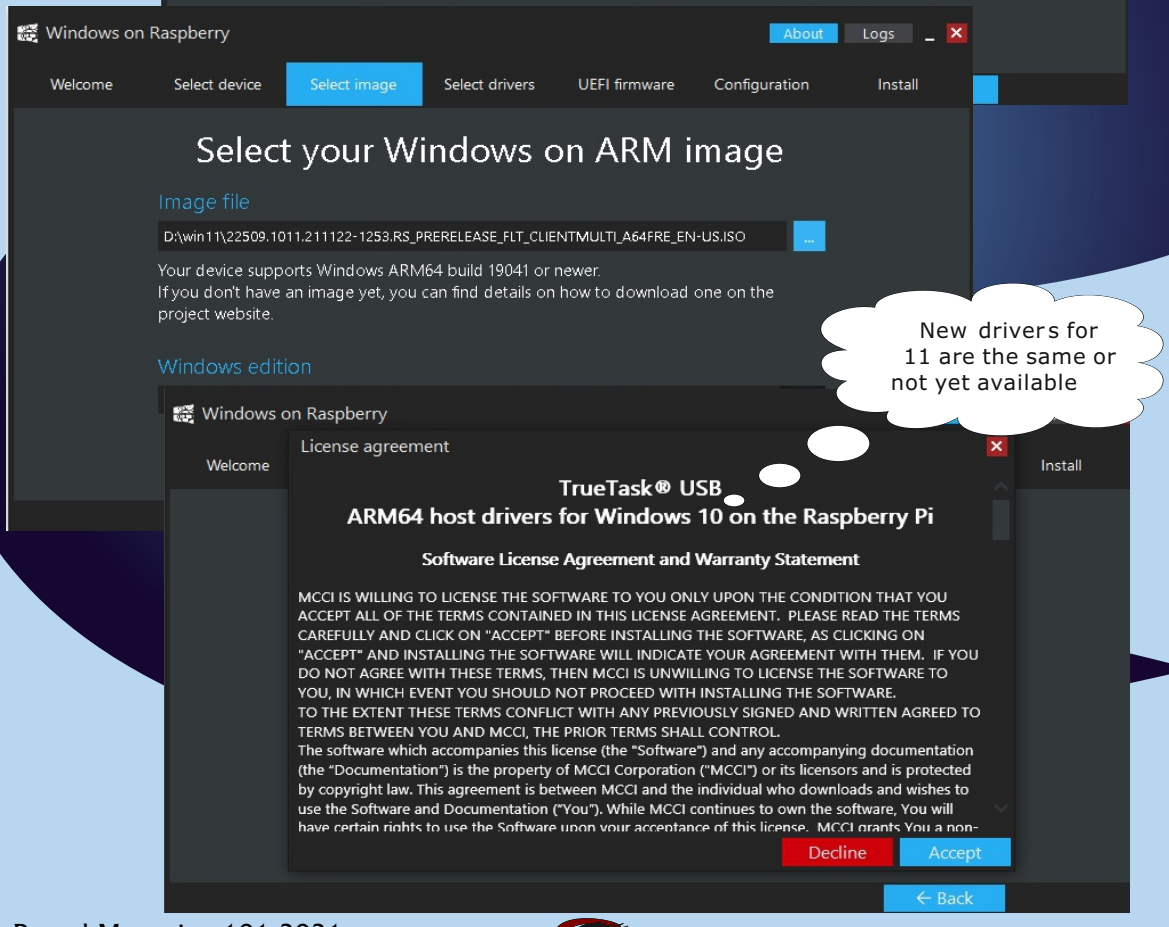


Figure 38:

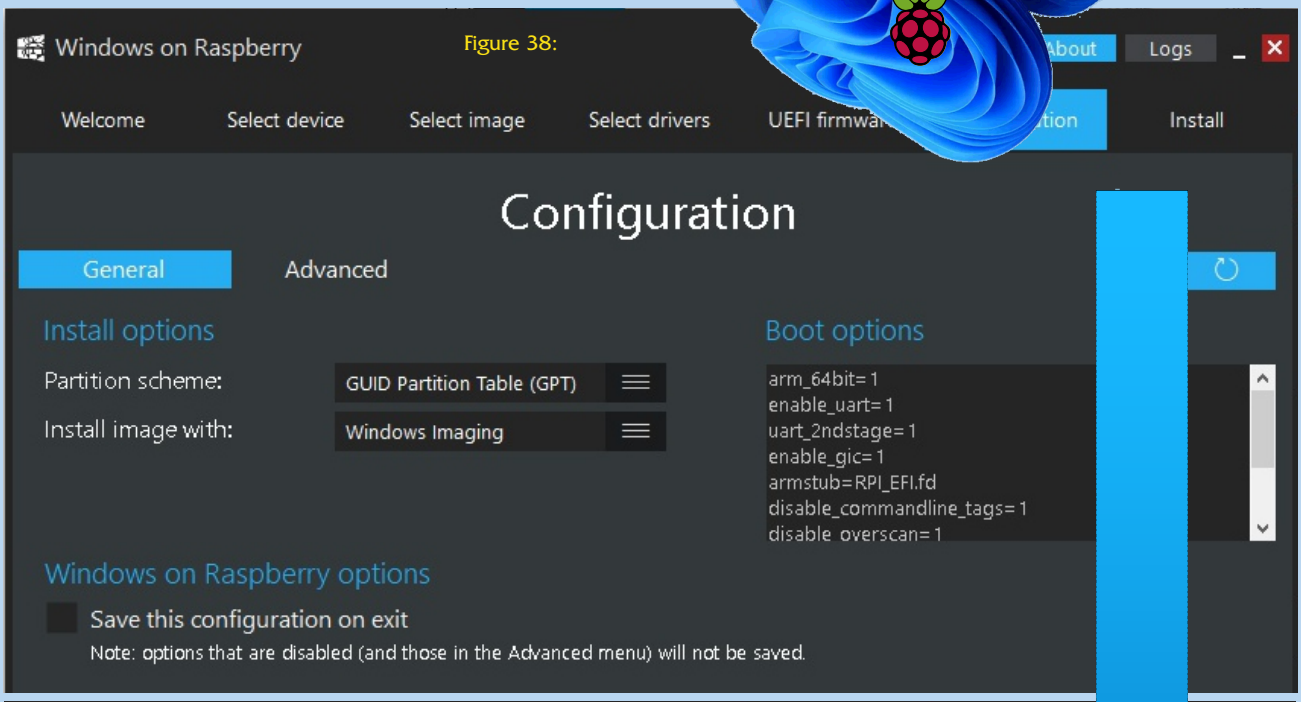


Figure 39:

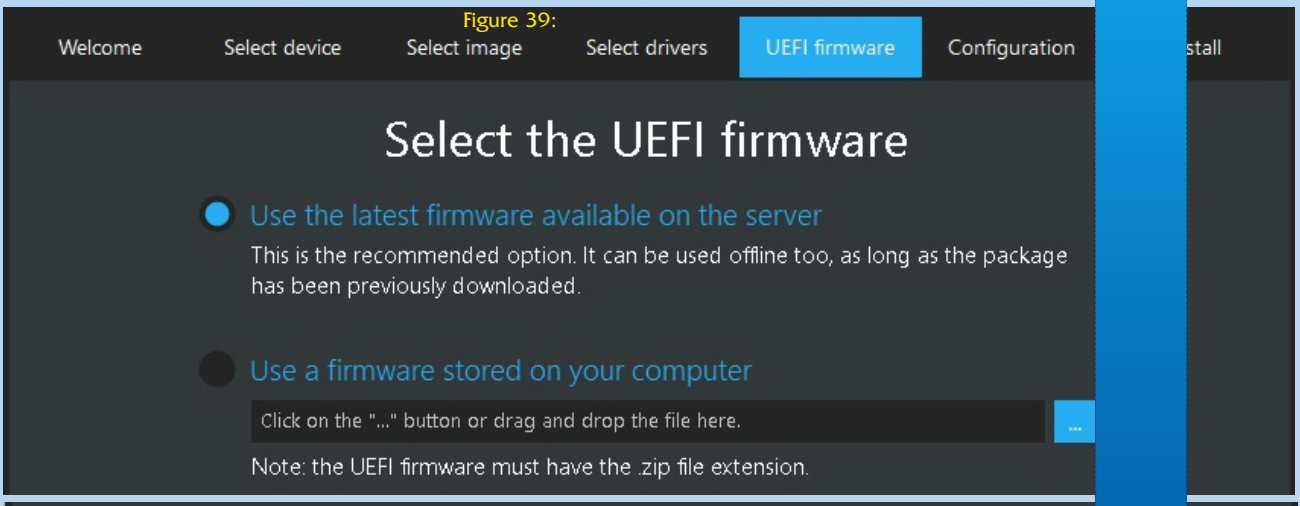


Figure 40:

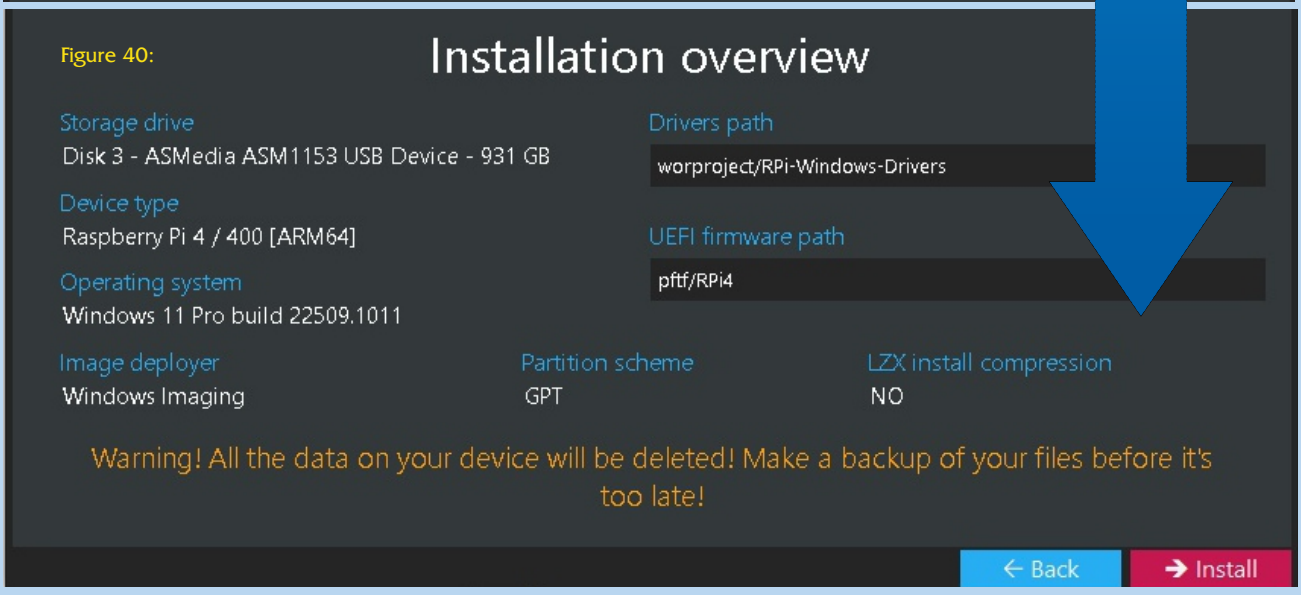


Figure 41:

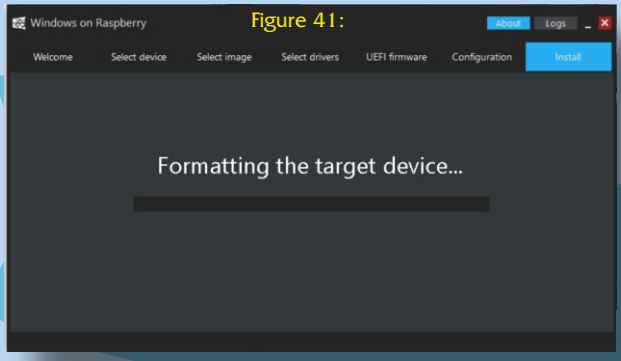


Figure 42:

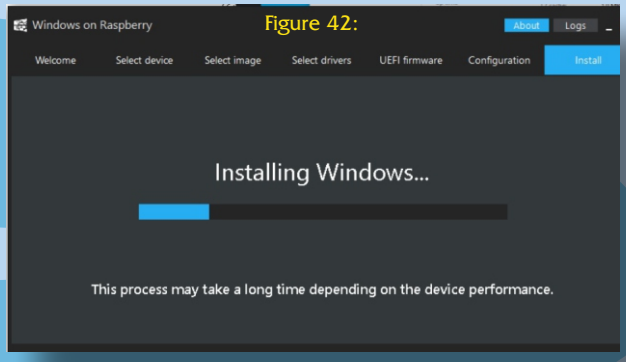


Figure 43:

Figure 44:

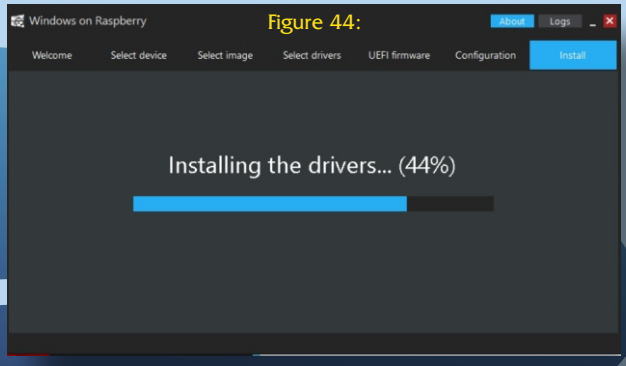


Figure 45:

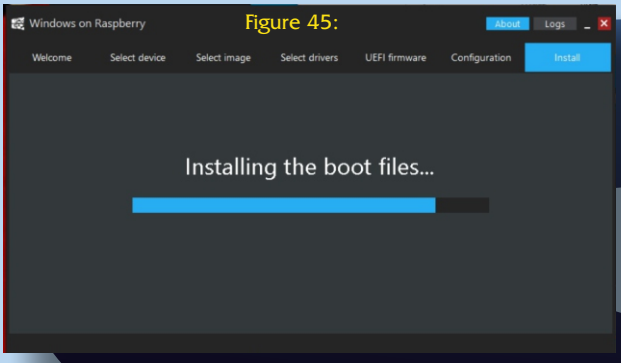


Figure 46:

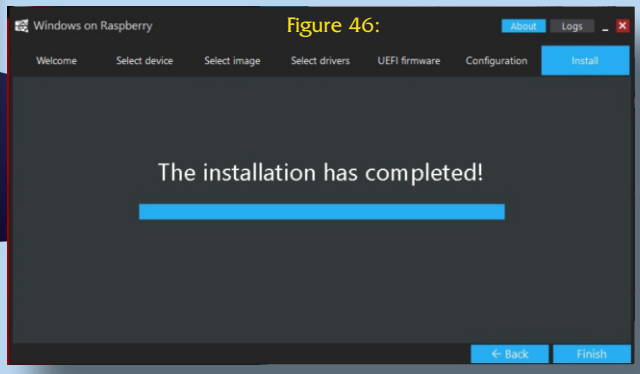


Figure 47: Starting Windows 11 on the the Raspberry Pi

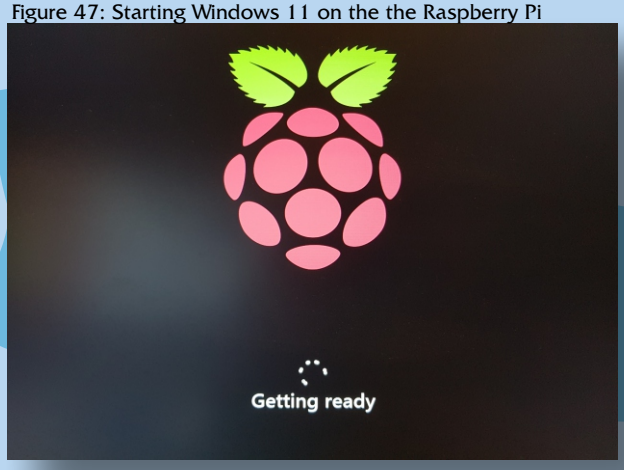


Figure 48: License Agreement

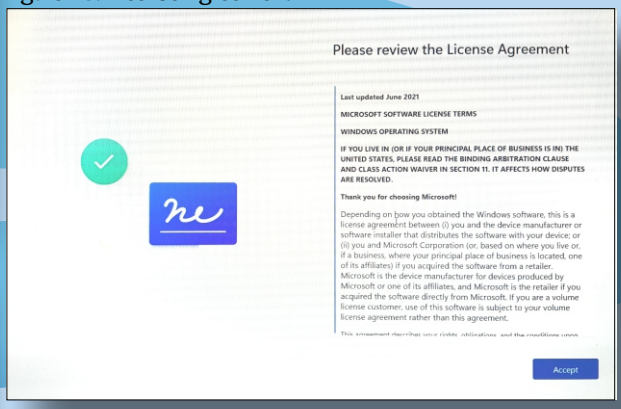


Figure 49: Country and Region

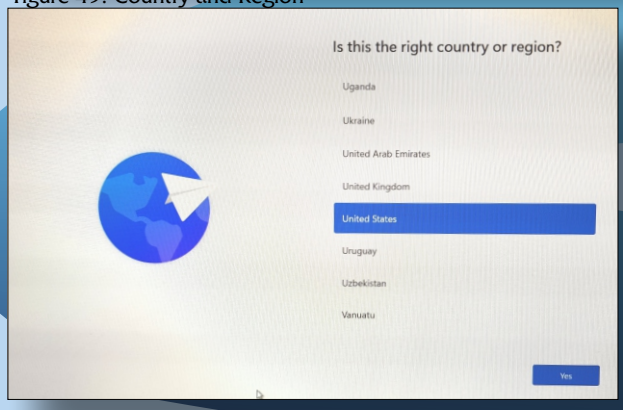


Figure 50: Choose the personal use or work or school

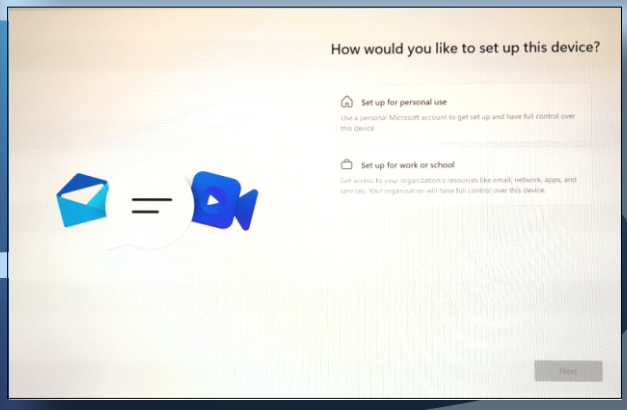


Figure 51: Updates

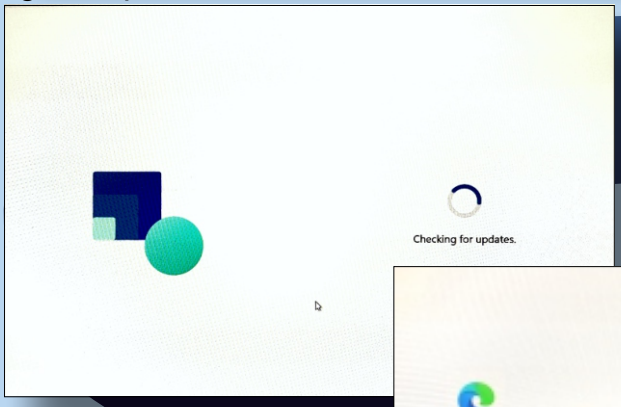


Figure 52 : adding the Microsoft account

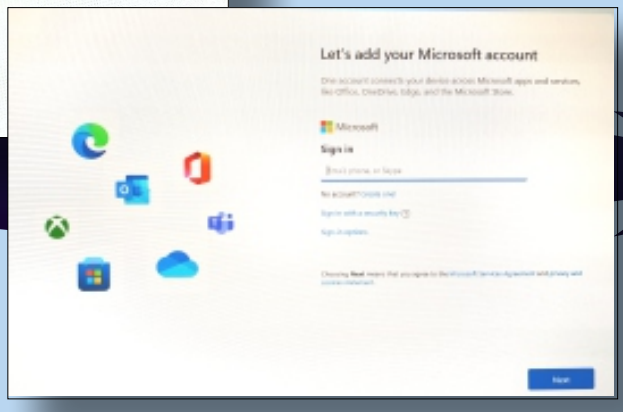
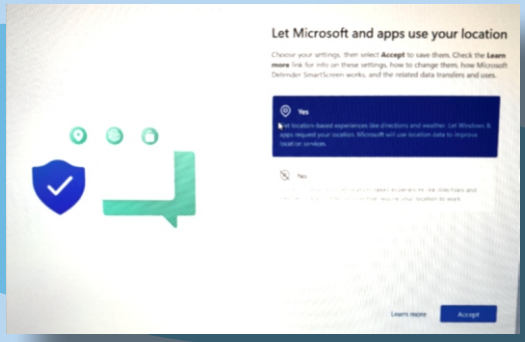


Figure 53: let Microsoft use location?



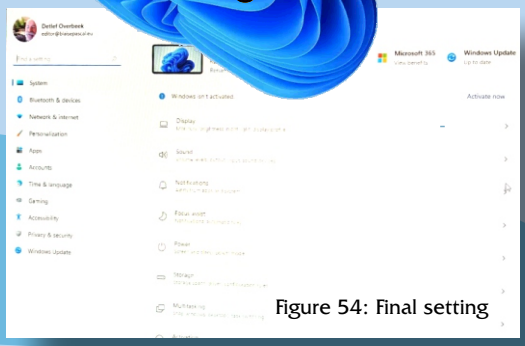


Figure 54: Final setting

Figure 55: Possibel apps to install t
hat are already available

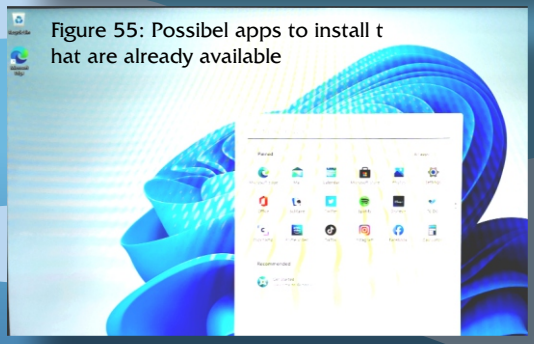
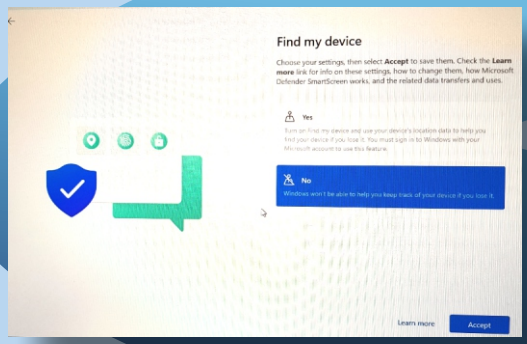


Figure 56: Finding your device if you loose it



From here on some screenhotsrus of
the installment of Lazarus and Ddelphi

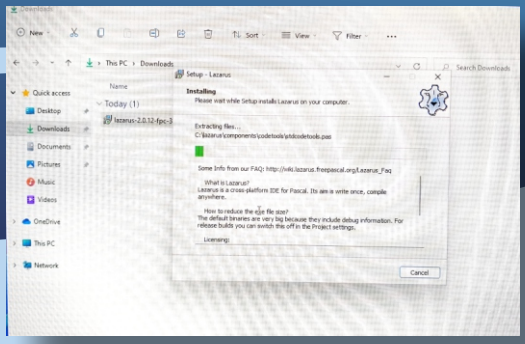
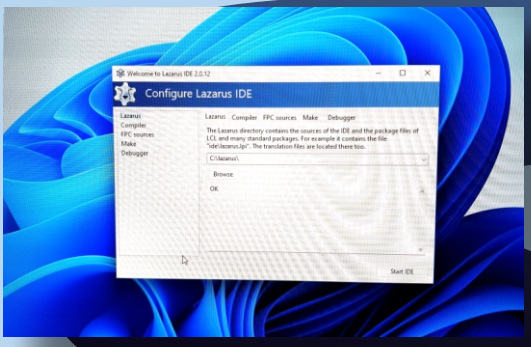


Figure 57: Installing Lazarus

Figure 58: Configuring and checking Lazarus installment



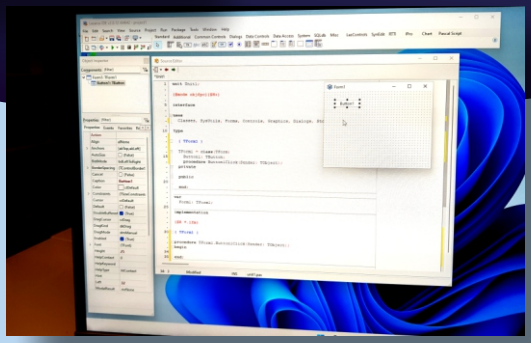
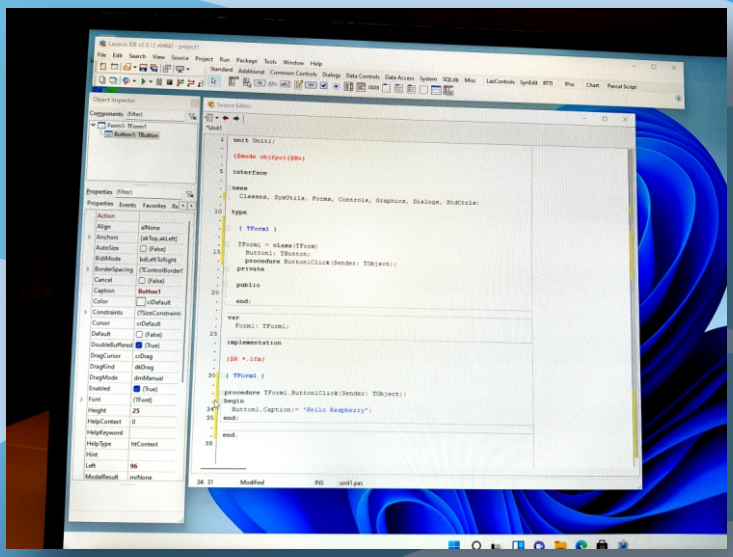


Figure 59:
Opening Lazarus for the first time

Figure 60: Lazarus is installed and ready for first use



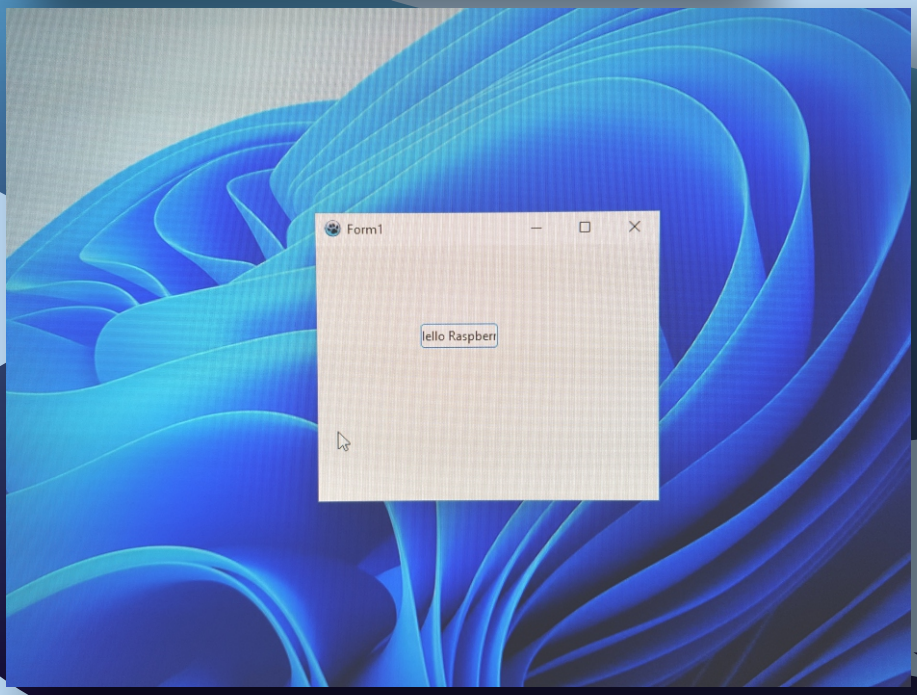


Figure 61: The first progam: Hello Raspberry

Figure 62: Starting Delphi 11

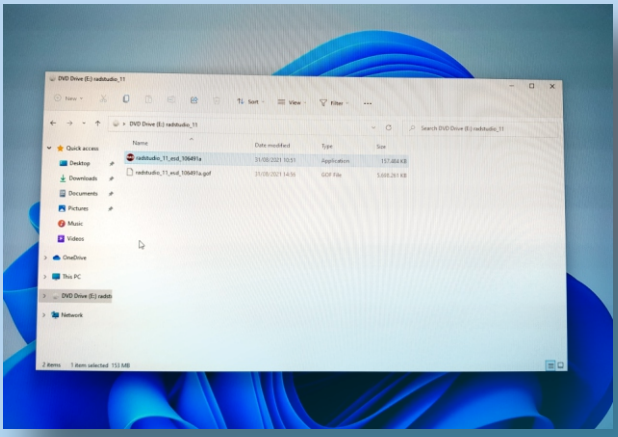


Figure 63: The first step

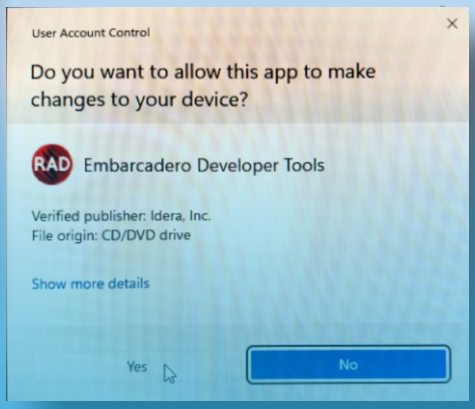


Figure 64: Platform Selection

Figure 65: The logo of Delphi appears...

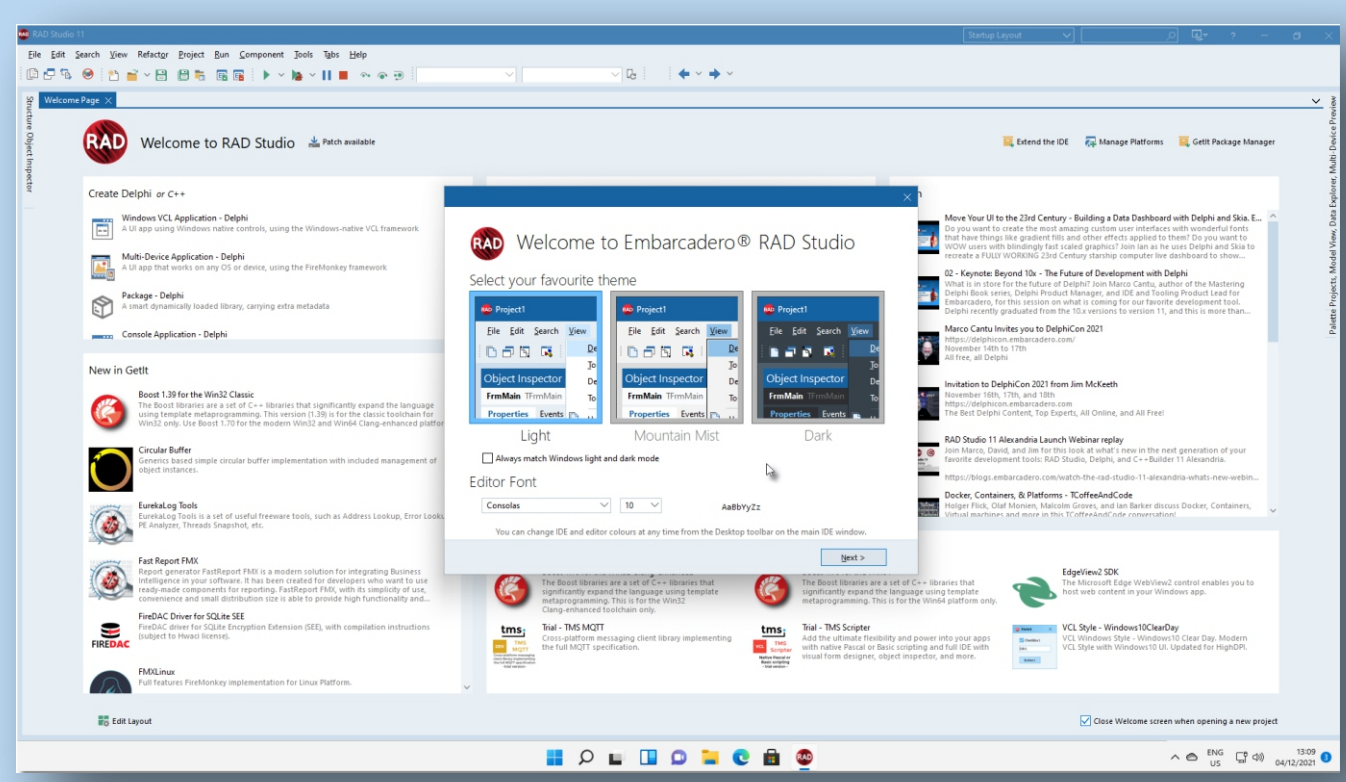
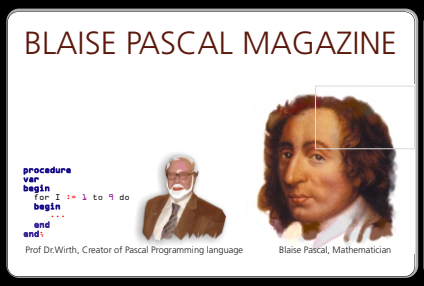
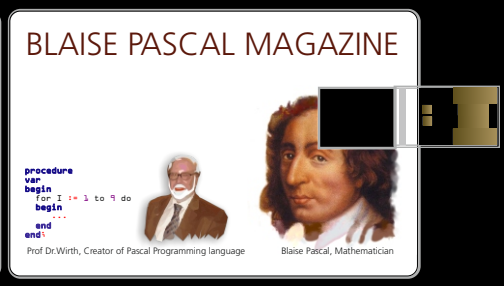


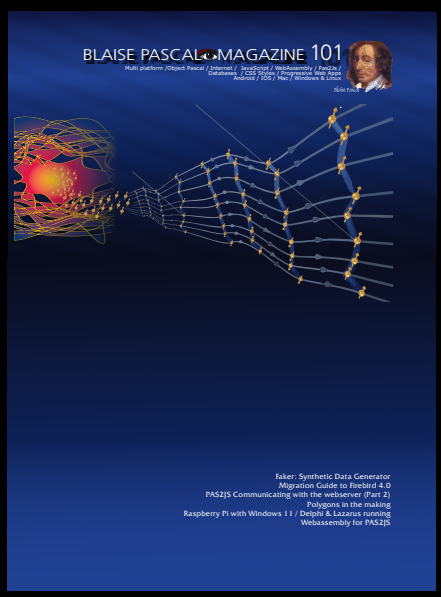
Figure 66: The last step Deelphi opens with its “Welcome Menu”

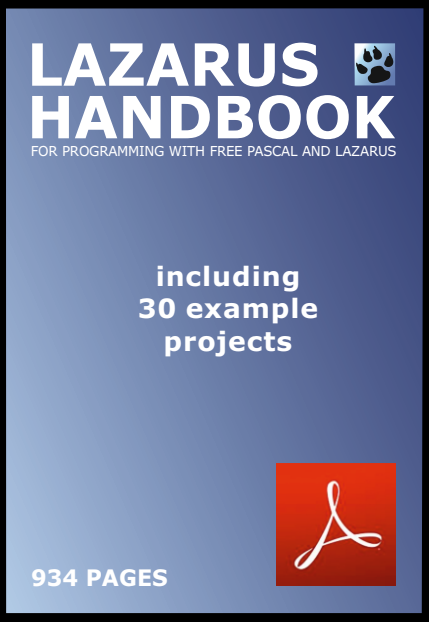


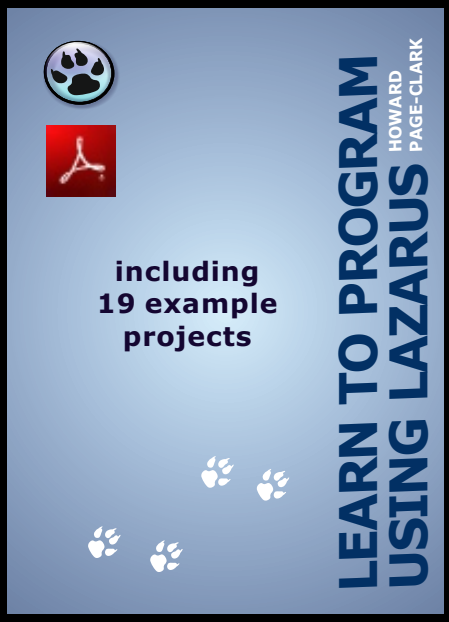


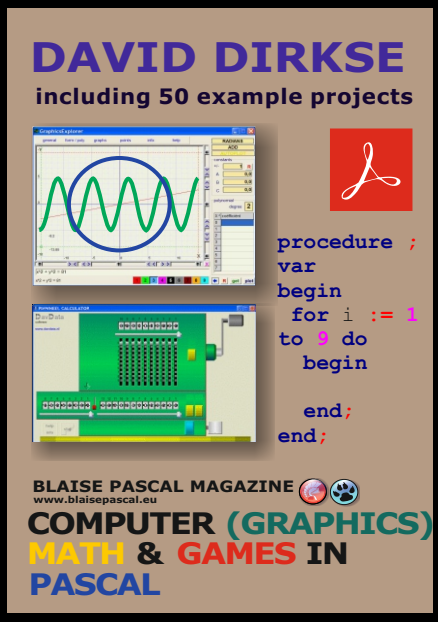
SUPER PACK 5 Items € 150 ex Vat

Normal Price € 280
75+60+50+35+50





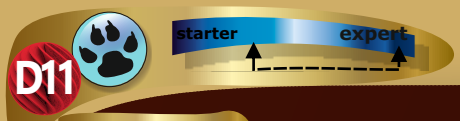




1. One year **Subscription**
2. **The newest LIB Stick**
   - including Credit Card USB stick
3. **Lazarus Handbook** - Personalized
   -PDF including Code
4. Book **Learn To Program** using Lazarus PDF including 19 lessons and projects
5. **Book Computer Graphics Math & Games** book + PDF including ±50 projects

https://www.blaisepascalmagazine.eu/product-category/special-offer/

By Michaël Van Canneyt



ABSTRACT

The Free Pascal and Lazarus foundation sponsored development of aWebAssembly backend for FPC. The backend is now usable in production, and we'll show how to work with it in this article

❶ INTRODUCTION

WebAssembly (Wasm) is gaining traction: Starting out as a way to make **Javascript** run faster in the browser (`Asm.js`), it has now become a full description of a runtime engine, designed to run bytecode in a safe way, regardless of where the code is running:
`https://webassembly.org/`

All Major browsers support the running of **WebAssembly** byte code, **Node.JS** and **Deno.** Not only that, but major languages (**C/C++,Rust, C#**) - can be compiled to **WebAssembly** using a special libc library, thus allowing a **C#, C/C+**+ program to run in the browser.

The developers at Mozilla took it even a step further: because **WebAssembly** is designed to be safe, sensitive parts of the browser are converted to **WebAssembly,** and then converted back to **C++**, thus guaranteeing that the resulting code is completely sandboxed and will not be able to penetrate into the rest of the browser.
A **webassembly** program can now be run in the browser, but also on a server, as part of **Javascript** runtimes such as **Node.JS** or **Deno,** or using a dedicated runtime:

wasmtime `https://wasmtime.dev/` ***** *is used creating the .exe file*
or **wasmer:** `https://wasmer.io/`

Both provide a command-line runtime engine that can load a **WebAssembly** file and run the code in it. They allow access to the filesystem and interaction with the console through a common **API** to allow the **WebAssembly** code to interact with the host environment. This **API** is called **WASI** (*which is an acronym for WebAssembly System Interface*):
`https://wasi.dev/`
Since some time, the Free Pascal compiler can emit **Webassembly** code, which also relies on the **WASI API** to talk to the host environment. The **WebAssembly** backend is meanwhile sufficiently mature to compile many of the packages and units supplied with **Free Pascal.**

The `Goto` statement is not yet implemented, but this is a matter of time before it is implemented. In this article, we explore how to make use of this new compiler backend.

❷ INSTALLATION

The **Free Pascal WebAssembly compiler** is not yet officially released. This means that you must build it yourself if you wish to use it. The **Free Pascal WebAssembly compiler** makes use of the **linker** of the **LLVM** project. So, the first step is to install the **LLVM** linker. The **LLVM linker** is part of **LLVM,** and can be downloaded here for **Windows:**

`https://github.com/llvm/llvm-project/releases/download/llvmorg-12.0.1/LLVM-12.0.1`

The installer will ask you if it must add the folder with binaries to the path: you must instruct it to do so. When it is done, you must copy the application `wasm-ld.exe` to `wasm32-wasi-wasm-ld.exe`, as the latter is what the compiler expects to find. For **Linux** and **MacOS,** the package manager can be used to install **llvm.** For example, on **Linux Ubuntu 20.04** this is done using:

```
apt install lld-12
 ln -sf /usr/lib/llvm-12/bin/wasm-ld ~/bin/wasm32-wasi-wasm-ld
```

For **MacOS,** the macports system can be used to install **llvm-12.** Obviously, you need to have the latest **Free Pascal** compiler installed. If you have the latest version of the **Lazarus IDE** installed, then you will have an up-to date compiler installed as well. The following commands assume that the **Free Pascal** compiler is installed on your system, and that the **fpc.exe** binary is in your `PATH`.

Using the installed compiler the **Free Pascal webassembly cross-compiler** must be built. This must be done with the latest sources of **FPC.** somewhere on your system, use git to clone the latest sources (*the following must be executed in a command-line window*):
git clone `https://gitlab.com/freepascal.org/fpc/source.git fpc`.
It shows a list where you can choose the operating system
When git has completed the clone operation, build the cross compiler. This can be done with the following commands:

```
cd fpc
make all OS_TARGET=wasi CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-" PP=fpc
cd compiler\utils
make all
cd ..\...
```

If all goes well, you will have built a `ppcrosswasm32.exe` compiler. This new compiler can be installed with the following command:

```
make install OS_TARGET=wasi CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-"
PP=fpc
cd compiler\utils
make install
cd ..\..
```

This will install a newer version of the fpc binary.
More detailed information on building and installing the **Free Pascal** compiler can be found on
`https://wiki.freepascal.org/Installing_the_Free_Pascal_Compiler`

```
c:\FPC>fpc -Twasi -Pwasm32 helloworld.pp
Free Pascal Compiler version 3.3.1 [2021/12/24] for wasm32
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: The WebAssembly System Interface (WASI)
Compiling helloworld.pp    (or helloworld.pas)
Linking helloworld.wasm
2 lines compiled, 0.1 sec

c:\FPC>_
```

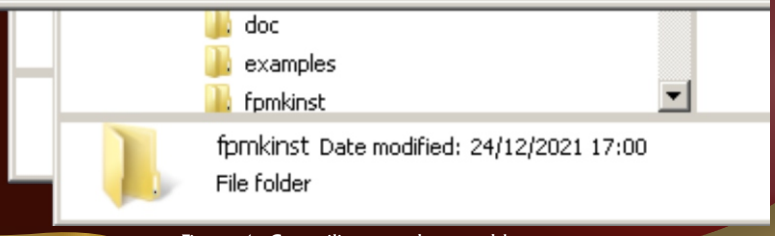


Figure 1: Compiling a webassembly program

❸ COMPILING FOR WEBASSEMBLY

Compiling with the **Free Pascal Webassembly Compiler** is not different from compiling for any other supported platform. We'll start with the simplest **Free Pascal** program, which we'll save somewhere in a file called `helloworld.pas`:

```
program helloworld;
begin
Writeln('Hello, world!');
end.
```

To compile this program from the command-line, the following can be done:

```
fpc -Twasi -Pwasm32 helloworld.pas
```

The compiler will compile and if all went well, you'll see some output as in figure 1 on page 3. Alternatively, the following completely equivalent command can be used:

```
ppcrosswasm32 helloworld.pas  (or helloworld.pp)
```

To compile for **WebAssembly** in **Lazarus,** there are several options, depending on which version of Lazarus you are using.

For all options, you must disable the generation of debug information in the **Project Options Dialog** under the page compiler **options - debugging.**
For the officially released version, there are 2 options to choose from. The first one is easiest, but has a drawback: In the **Tools-Options** dialog, select the `ppcrosswasm32.exe` from the following directory:

```
C:\FPC\3.2.2\bin\i386-Win32
```

This is shown in figure 2 on page 4.
After doing this, every project you compile will be compiled for WebAssembly.
(*and that includes the IDE itself if you decide to rebuild it*)
Obviously this is normally not desirable, in practice only certain projects will be compilable for **WebAssembly.** The better way is to use the Compile commands from the **project options**, as shown in figure 3 on page 5.

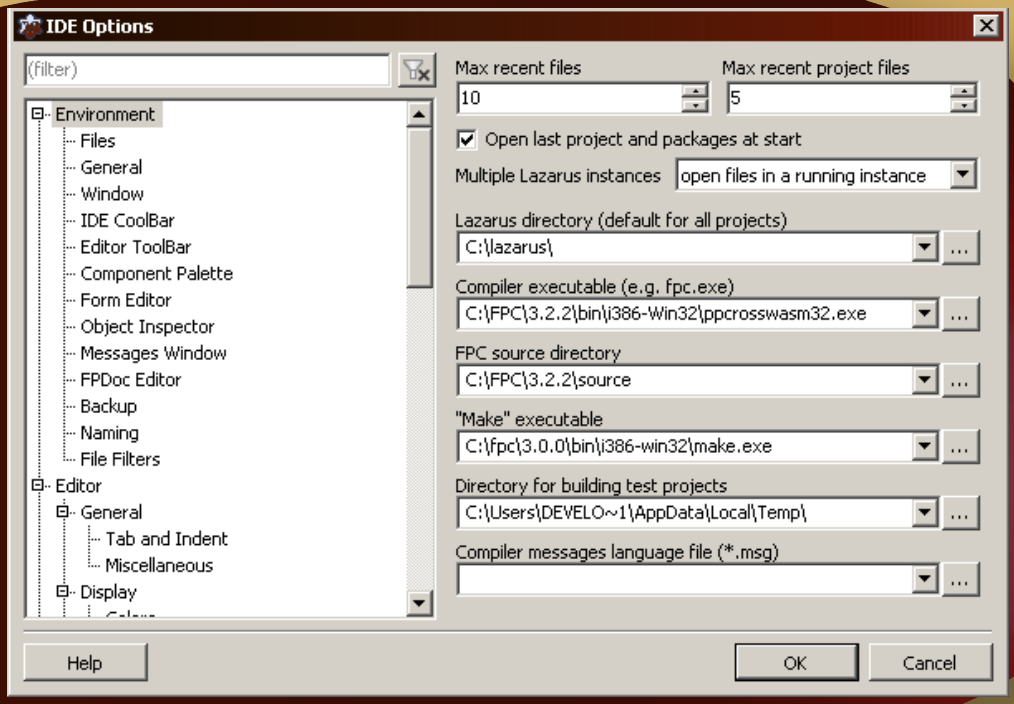


Figure 2: Selecting the webassembly compiler

The `Execute Before` command can be used to run the cross-compiler. For this all options after **Call** on must be set, and the command must be set to

```
C:\FPC\3.2.2\bin\i386-Win32\ppcrosswasm32 $(ProjFile)
```

You can add any other command-line options that you wish to have. Under Parsers, select **FPC.** This tells the **IDE** to parse the output of the command as it would parse **FPC** output. Then, under the **Compiler** section, disable all the **CALL on** options.
After this, when you compile, it will be as if you compile a program for the native OS on which the IDE is running, see figure 4 on page 5.
If you are using the development version of Lazarus, the above options will still work. However, with the development version it is even easier to compile for webassembly with the development version. It is sufficient to select wasm32 as the target processor, and wasi as the target OS, as shown in figure 5 on page 5. The compiled file will also have the correct extension (`.wasm`) for **Lazarus** sources of `December 28 2021` or later. Or version of **Lazarus 2.2.0**

❹ RUNNING A WEBASSEMBLY PROGRAM NATIVELY

Now that we've succesfully compiled a simple webassembly program, we of course will want to run it. For this, we can use the wasmer or wasmtime command-line **WebAssembly** runtimes. The runtime command can be downloaded from:

```
https://github.com/bytecodealliance/wasmtime/releases/tag/v0.32.0
```

Once installed, running the generated webassembly is easy. In a command-line terminal, run the following command in the **WEBASSEMBLY** project directory:

```
wasmtime helloworld.wasm
```

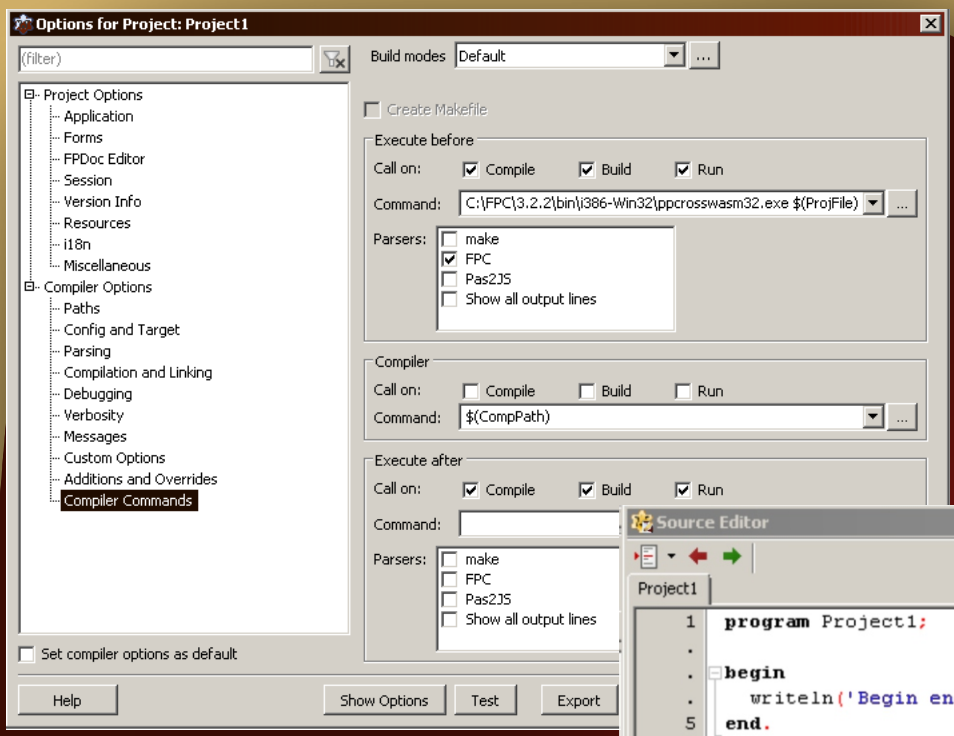


Figure 3: Using the webassembly compiler for a single project

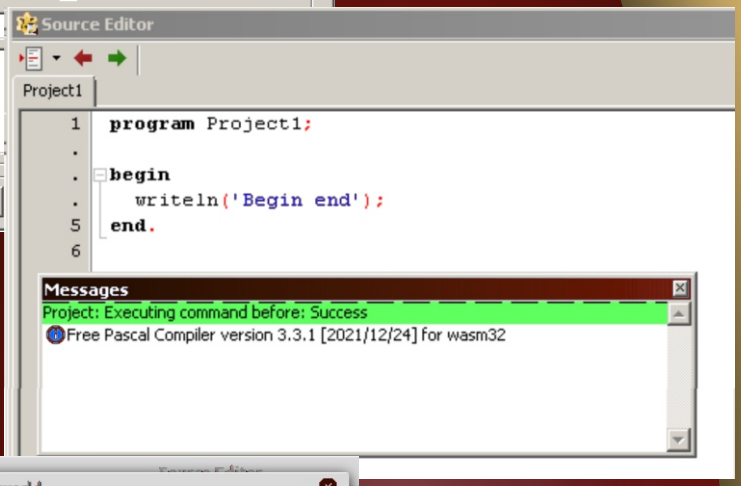


Figure 4: Compilation with the webassembly compiler

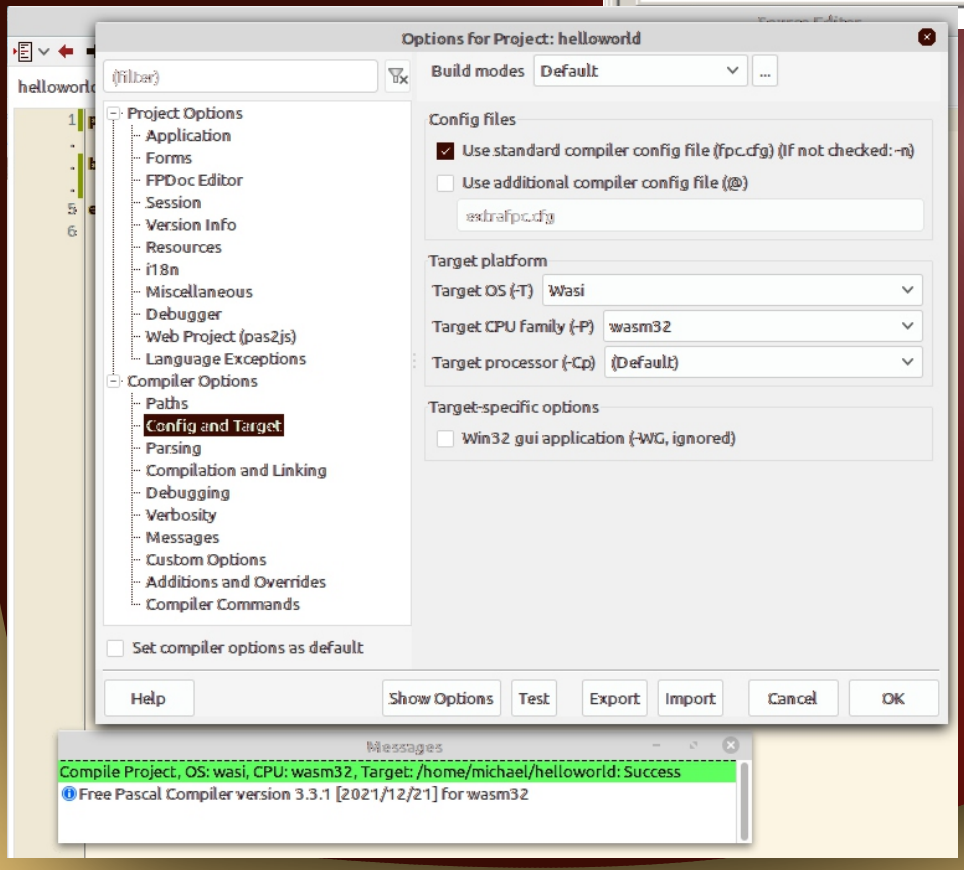


Figure 5: Using the webassembly compiler for a single project

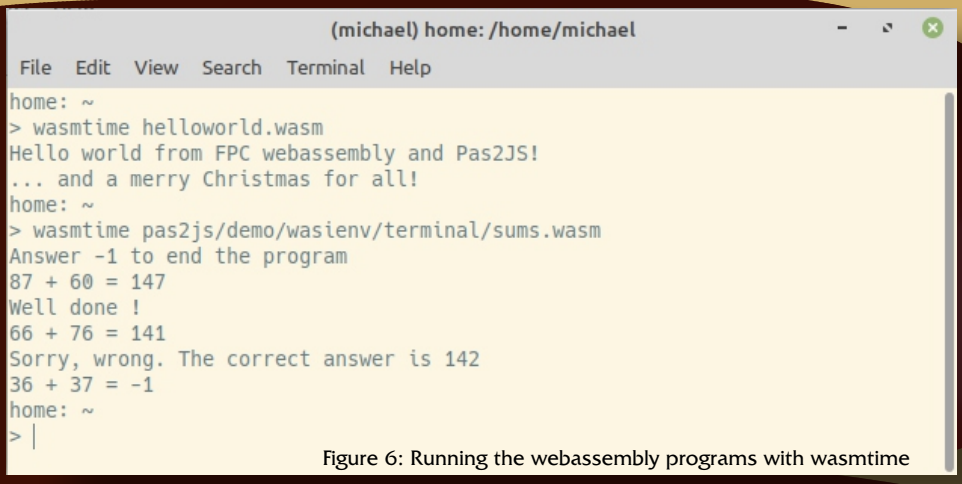


Figure 6: Running the webassembly programs with wasmtime

Or, you can compile and run the `sums.pp` demo project, which is part of the **pas2js** demos for **Webassembly** (*you can find it in the folder* `demos/wasienv/terminal`):

```
ppcrosswasm32 sums.pp
wasmtime sums.wasm
```

And the result will look like figure 6 on page 6, where you can see that the sums program actually reads input from the terminal.

❺ RUNNING A WEBASSEMBLY PROGRAM IN THE BROWSER

A **webassembly** program can be loaded and run in the browser. The browser offers **APIs** to do so, and using Pas2JS, you can easily create a hosting environment for your **webassembly** program. There are currently 2 options to do so:

❶ Manually load and run the webassemly file using the provided WASI environment class `TPas2JSWASIEnvironment`.

❷ Use the Pas2js-provided TWASIHostApplication application class and let it do the heavy lifting for you – it uses the `TPas2JSWASIEnvironment` class in the background.

We'll start with the former method. Let's start by explaining what the `TPas2JSWASIEnvironment` class is for: **WebAssembly** standards do not make any assumptions about the environment in which the **WebAssembly** code is executed. Yet, **WebAssembly** would not be interesting if it could not interact with the environment.

To interact with the outside world, **WebAssembly** code relies on imported routines: the specifications do point out the mechanism to call external routines.
The **FPC RTL** for **WebAssembly** currently follows the **WASI** standard to interact with the host environment. The **WASI** standard describes a minimal set of import routines, and is used by the **WasmTime** and **Wasmer** runtime environments.
The `TPas2JSWASIEnvironment` class is implemented in `Pas2JS`, and offers all the callbacks needed for the **WebAssembly** runtime generated by **FPC:** These are the callbacks specified by the **WASI** standard. Although all callbacks are present, they are currently not all implemented. The class offers also the possibility to hook additional **APIs** and catch input and output. The following is the public **API** of this class:

```
TPas2JSWASIEnvironment = class(TObject)
  Function GetUTF8StringFromMem(aLoc,aLen:Longint):String;
  Procedure AddImports(aObject:TJSObject);
  Property ImportObject:TJSObject;
  Property IsLittleEndian:Boolean;
  Property OnStdOutputWrite:TWASIWriteEvent;
  Property OnStdErrorWrite:TWASIWriteEvent;
  Property OnGetConsoleInputBuffer:TGetConsoleInputBufferEvent;
  Property OnGetConsoleInputString:TGetConsoleInputStringEvent;
  Property Instance:TJSWebAssemblyInstance;
  Property Exitcode:Nativeint;
  // Default is set to the one expected by FPC runtime:
  // wasi_snapshot_preview1
  Property WASIImportName:String;
end;
```

The `GetUTF8StringFromMem` method is a utility call that will retrieve an `UTF8` string from the **WebAssembly** memory (indicated by a location and length), and returns it as a **Javascript** string. The `AddImports` call will add the **WASI** imports to the passed object, as well as any additional **APIs** you have defined (more about that later).

The following properties are also available:

`IsLittleEndian`	A property describing whether the **WebAssembly** memory is little-endian or big-endian.
`OnStdOutputWrite`	Called when the **WebAssembly** program writes to standard output.
`OnStdErrorWrite`	Called when the **WebAssembly** program writes to standard error.
`OnGetConsoleInputBuffer`	Called when the **WebAssembly** program tries to read from standard input. Use this event if you wish to pass binary data.
`OnGetConsoleInputString`	Called when the **WebAssembly** program tries to read from standard input. Use this event if you wish to pass textual data.
`Instance`	This is the currently running `TJSWebAssemblyInstance`.
`ExitCode`	This is the exit code of the **WebAssembly** program.
`WASIImportName`	This is the name for the import object for the **WASI API**: the default is `wasi_snapshot_preview1`.

So, how to use this class to run a webassembly file ?

To demonstrate this, we create a small Pas2JS program in the Lazarus IDE (see the article on writing real-world Pas2JS applications on how to get started with Pas2JS), and we instruct the IDE to use the TBrowserApplication for the program source.

In the application class' constructor, we create

```
constructor TMyApplication.Create(aOwner:TComponent);
begin
  inherited Create(aOwner);
  FWasiEnv:=TPas2JSWASIEnvironment.Create;FWasiEnv.OnStdErrorWrite:=@DoWrite;
  FWasiEnv.OnStdOutputWrite:=@DoWrite;
end;

procedure TMyApplication.DoWrite(Sender:TObject;const aOutput:String);
begin
  Writeln(aOutput);
end;
```

As you can see, we use the Pascal `Writeln` function to write the standard&error output of the webassembly program. Because we're using the `BrowserConsole` unit, the output will be written in the **HTML** page. What was created in the constructor must be destroyed in the destructor, so we implement that too:

```
destructor TMyApplication.Destroy;
begin
  FreeAndNil(FWasiEnv);
  inherited Destroy;
end;
```

The `DoRun` method of the application object must be overriden to implement the actual program logic. In our case, we simply call `InitWebAssembly`:

```
procedure TMyApplication.doRun;
begin
Terminate;
InitWebAssembly;
end;
```

The `InitWebAssembly` method is where we set up the **WebAssembly** environment. The environment for a **WebAssembly** program is simply a **Javascript** object that contains various configuration objects as well as routines to be imported in the **WebAssembly** runtime. You can provide more routines than the environment needs, but all routines that the environment needs must be present in the import object. Two important (*but optional*) objects in this regard are:

- The `TJSWebAssemblyMemory` object with memory that can be made available to the **webassembly** runtime.
- a `TJSWebAssemblyTable` object may be specified that will contain a list of callable functions (*or imported functions*): These are functions that are defined in the **WebAssembly** module, and which can be called directly from **Javascript.**

The memory object takes a descriptor record for the constructor. This descriptor specifies the initial and maximum memory for the **WebAssembly** memory object. The values are specified in **WebAssembly** pages with 64Kb size. Similarly, the table uses a descriptor which allows to set initial and maximum sizes for the table, and what table you want: the 'anyfunc' value tells the **WebAssembly** engine to fill the table with all available functions.

```
procedure TMyApplication.InitWebAssembly;
Var  mDesc : TJSWebAssemblyMemoryDescriptor; tDesc: JSWebAssemblyTableDescriptor;
  ImportObj : TJSObject;
begin
  // Setup memory
  mDesc.initial:=256;
  mDesc.maximum:=256;
  FMemory:=TJSWebAssemblyMemory.New(mDesc);
  // Setup table
  tDesc.initial:=0;
  tDesc.maximum:=0;
  tDesc.element:='anyfunc';
  FTable:=TJSWebAssemblyTable.New(tDesc);
  // Setup ImportObject
  ImportObj:=new([
    'js', new([
      'mem', FMemory,
      ,tbl', FTable
    ])
  ]);
  FWasiEnv.AddImports(ImportObj);
  CreateWebAssembly('helloworld.wasm',ImportObj)._then(@initEnv)
end;
```

Note that the current implementation of **FPC WebAssembly** does not import the `js.mem` or `js.tbl` memory objects, but you can import them manually, so the above is just for demonstration purposes in case you wish to use additional memory.
The important call here is to the `FWasiEnv.AddImports` method: this method will add all necessary **WASI** and additional optional exports to the **WebAssembly** import object. After the call to `AddImports`, the `ImportObject` object is ready to be used in the `CreateWebassembly` call: This call returns a promise, which will result in a `TJSInstantiateResult` object. We let the promise resolve in the `InitEnv` method:

```
function TMyApplication.InitEnv(aValue: JSValue): JSValue;
Var Module : TJSInstantiateResult absolute aValue; exps : TWASIExports;
begin
  Result:=True;
  Exps := TWASIExports(TJSObject(Module.Instance.exports_));
  FWasiEnv.Instance:=Module.Instance;
  Exps.Start;
end;
```

The `Module` variable is just a declaration to avoid typecasts. The `TWASIExports` class is an extension of the `TJSModulesExports` class: this class exports the memory of the **WebAssembly** object, and contains the `Start` symbol. The `Start` symbol is the name of the program entry point, the only symbol the **FPC** runtime exports by default.

With this definition, it will be clear that the `Exps.Start` statement actually calls the program's main pascal function (*the begin of the program*). It's important to realize that this function does not return as long as the **WebAssembly** program is running, thus potentially blocking the browser.

The `CreateWebAssembly` call loads the **wasm** file, and calls all the necessary **WebAssembly** functions to compile and instantiate the **WebAssembly** instance:

```
function TMyApplication.CreateWebAssembly(Path: string; ImportObject: TJSObject): TJSPromise;
begin
  Result:=window.fetch(Path)._then(
    Function (res : jsValue) : JSValue
      begin
        Result:=TJSResponse(Res).arrayBuffer._then(
          Function (res2 : jsValue) : JSValue
            begin
              Result:=TJSWebAssembly.instantiate(TJSArrayBuffer(res2), ImportObject);
            end,
          Nil)
        end,
        Nil);
end;
```

The following happens:

1. The `Fetch` call will fetch the **webassembly** file, and retuns a promise.
2. The promise resolves to a `TJSResponse` result, and this is converted to an `JSArrayBuffer` – this conversion is again returning a promise.
3. The converted array buffer contains the **webassembly** bytecode wich is then passed to the `TJSWebAssembly.instantiate` function which creates a **WebAssembly** runtime instance.

The result of the `CreateWebAssembly` function is a promise, which resolves to the result of the `TJSWebAssembly.instantiate` function (*a promise in itself*). When the instantiated **WebAssembly** runtime instance is ready, the function resolves to a `TJSInstantiateResult` instance.

A shorter (and faster) version of this call is:

```
function TMyApplication.CreateWebAssembly(Path: string; ImportObject: TJSObject): TJSPromise;
begin
  Result:=TJSWebAssembly.instantiateStreaming(Fetch(Path), ImportObject);
end;
```

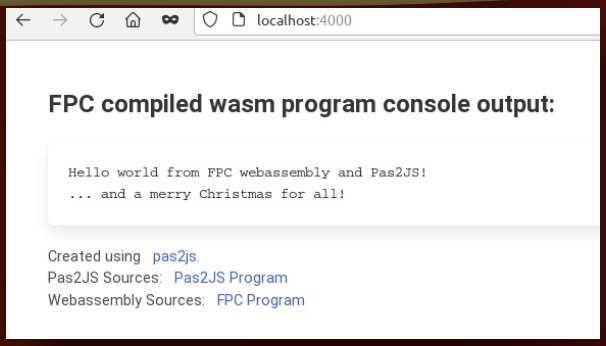


Figure 7: Running the webassembly program in the browser

The difference between `InstantiateStreaming` and `Instantiate` calls is that the former starts compiling the **WebAssembly** as the bytes come in from the fetch operation. However, you may wish to instantiate a **WebAssembly** runtime multiple times: in that case it may be better to keep the **WebAssembly** in memory as soon as it is loaded.

We can add some **HTML** to beautify the page in which this program is embedded, and after the **PAS2JS** program has finished running, this leads to a page such as can be seen figure 7 on page 10. Since the program is loaded as soon as the **HTML** page is loaded, this is also the initial view of the page. You can easily check this by adding a button to the page, and use its `OnClick` event to call the `InitWebAssembly` function.

We mentioned earlier that there are 2 ways to run your **WebAssembly** program. As all long-time users know, **PAS2JS** and the **Lazarus IDE** continuously try to make things easier for the developer. That is why **PAS2JS** provides the `TWASIHostApplication` object:
a descendent of `TBrowserApplication`. In the development version of **Lazarus,** the **New Project** dialog's **Web Browser Application** entry has an additional option called **Host WebAssembly** program (*see figure 8 on page 13*): When checked, you can enter the **URL** of the **wasm** file you wish to load. In that case, the new project wizard generates the following code, which makes use of the `TWASIHostApplication` class:

```
program Project1;

{$mode objfpc}

uses browserapp, wasihostapp, JS, Classes, SysUtils, Web;

type
  TMyApplication = class(TWASIHostApplication)
    procedure doRun; override;
  end;

procedure TMyApplication.doRun;
  begin
    StartWebAssembly('helloworld.wasm');
    Terminate;
  end;

var Application : TMyApplication;
begin
  Application:=TMyApplication.Create(nil);
  Application.Initialize;
  Application.Run;
end.
```

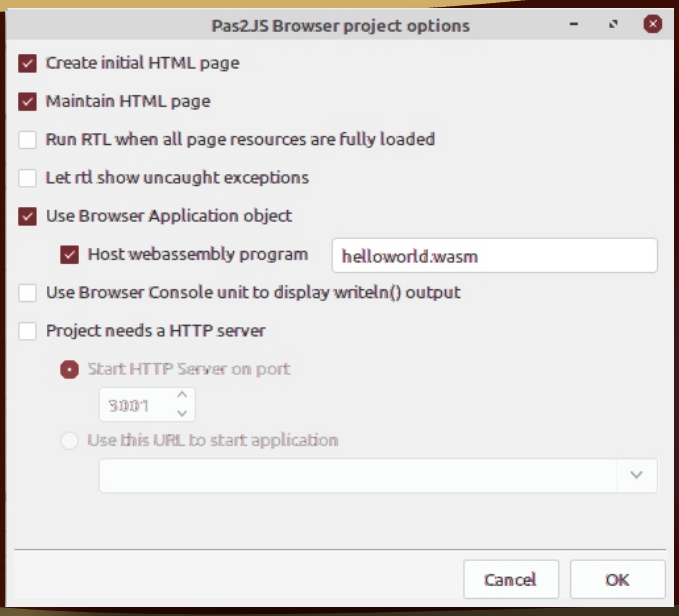


Figure 8: The enhanced new pas2js program dialog

This project is ready to run: on startup it will use the `StartWebAssembly` method to load the `helloworld.wasm` webassembly in a standard `TPas2JSWASIEnvironment` environment. This environment writes output to the browser console, and input is obtained using the Prompt call of the window (*a blocking call*). The environment can be modified using properties of the application object.

❻ EXTENDING THE WEBASSEMBLY ENVIRONMENT

As mentioned before, the **WebAssembly** specification does not contain an **API** for interacting with the outside world. In the context of the browser, this means that there is no standard **API** for accessing the **DOM** and changing the web page. However, the specification does describe how to import functions. This mechanism is used in the **WASI** specification to enable low-level access to the host environment: The **API** caters mainly for file and directory access.

So, if we want to manipulate the webpage in the browser, we'll have to provide an API to the **WebAssembly** environment. The **API** can be anything we want: we have complete control over what we allow the **WebAssembly** environment to do. The `TJSWasiEnvironment` class has support for easily adding additional **APIs** to the webassembly environment.

This support comes in the form of the `TImportExtension` class. This class serves as a parent class for classes to extend the standard **WASI** environment. It has the following public declaration:

```
TImportExtension = class (TObject)
Public
  Constructor Create(aEnv : TPas2JSWASIEnvironment); virtual;
  Procedure FillImportObject(aObject : TJSObject); virtual; abstract;
  Function ImportName : String; virtual; abstract;
  Property Env : TPas2JSWASIEnvironment Read FEnv;
end;
```

The constructor has a single argument: the `TPas2JSWASIEnvironment` instance which must be extended; The `TImportExtension` class will register itself in the environment. When the AddImports method of the `TPas2JSWASIEnvironment` instance is called to initialize the imports for the webassembly instance, all registered `TImportExtension` classes will be asked to create an import object, which will be added to the import object passed to the **WebAssembly** instance. The environment is available later in the `Env` property, this will allow the implementation to access the instance.

There are two abstract functions which must be implemented by a descendent:

FillImportObject this method must add all import methods to the `aObject` parameter; Note that this object is not the **WASI** enviroment which is extended: each extension object will be imported with a unique object.

ImportName this is the name that is used to add the object passed in **FillImportObject** to the global **WebAssembly** impory object. Now we know how to pass additional functions to a **WebAssembly** runtime. But how must the code running in the **WebAssembly** engine import such a function? Well, this happens in exactly the same manner as one would import a function from an external library.

Let's analyse the following call, which is part of the **FPC RTL**, and is used in the system unit:

```
function __wasi_clock_res_get(
id: __wasi_clockid_t;
resolution: P__wasi_timestamp_t
): __wasi_errno_t; external 'wasi_snapshot_preview1' name 'clock_res_get';
```

This declares a function `__wasi_clock_res_get` which accepts 2 arguments, an ID and a pointer. The key elements here are the `external` and name modifiers:

- The external specifies the name of the import object in which to find the function (in this case wasi_snapshot_preview1).
- The name is the name of the function that must be present in the object.

In the `TPas2JSWASIEnvironment` class we find a function called `clock_res_get`:

```
function clock_res_get(clockId, resolution: NativeInt): NativeInt; virtual;
```

Note that the ID (an integer) and resolution (a pointer) are both converted to an integer: the reason is that every address is just an index in the global memory array of the **WebAssembly** engine.

Seeing that the name of the method is the correct name expected by the **WebAssembly** runtime, does this mean we can simply attach the `TPas2JSWASIEnvironment` instance to the import object? Unfortunately not.

When the **webassembly** code calls this function, the `this` variable (*known in pascal as Self*) is empty. That is a problem because all methods (*static methods excepted*) of a class expect a `Self` pointer.

So we must register a function that does supply the `this`.

Fortunately, this is easy. Like all extensions, the functions that `TPas2JSWASIEnvironment` exposes are

```
procedure TPas2JSWASIEnvironment.GetImports(aImports: TJSObject);
begin
aImports['args_get']:=@args_get;
aImports['args_sizes_get']:=@args_sizes_get;
aImports['clock_res_get']:=@clock_res_get;
// ...
end;
```

As you can see, the `clock_res_get` is attached to the aImports objects with the correct name. The @ operator will bind `this` to the actual function in the object, so when `clock_res_get` is called, Self will be available. Note that because of this, the function name must not necessarily equal the name used in the **WebAssembly** runtime: the name can always be corrected in the `GetImports` call.

To demonstrate how this can be used, we'll add the possibility to let the webassembly draw on a **HTML** canvas. The first thing to do is to create the import functions. We create a unit for this, we'll call it **WebCanvas.** The following is part of the unit:

```
unit webcanvas;

interface

Type
  TCanvasError = longint;
  TCanvasID = longint;
  PCanvasID = ^TCanvasID;

Const
  ECANVAS_SUCCESS = 0;
  ECANVAS_NOCANVAS = 1;
  ECANVAS_UNSPECIFIED = -1;

function __webcanvas_allocate(
  SizeX : Longint;
  SIzeY : Longint;
  aID: PCanvasID): TCanvasError; external 'web_canvas' name 'allocate';

function __webcanvas_moveto( aID : TCanvasID;
  X : Longint;
  Y : Longint): TCanvasError; external 'web_canvas' name 'moveto';

function __webcanvas_filltext( aID : TCanvasID;
  X : Longint;
  Y : Longint;
  aText : PByte;
  aTextLen : Longint): TCanvasError; external 'web_canvas' name 'filltext';

  // ...

implementation

end.
```

As you can see, there is no implementation for these methods: the implementation will be imported from the **Javascript** host environment. From the declarations, you can see that these methods must be part of an import object called web_canvas. We can use this to create a canvas class that can be used in the webassembly runtime:

```
TWebCanvas = class(TObject)
private
  FCanvasID : Longint;
  FHeight: Longint;
  FWidth: Longint;
Protected
  Procedure Check(aError : TCanvasError; const aMsg : String = '');
Public
  Constructor Create(aWidth,aHeight : Longint);
  Procedure moveto(X : Longint;Y : Longint);
  Procedure FillText(X : Longint;Y : Longint;S : UTF8String);
end;
```

The Check method serves to check the return of the imported functions: all functions return - an arbitrary convention - an error code. The `Check` function converts this to an exception:

```
procedure TWebCanvas.Check(aError: TCanvasError; const aMsg: String);
begin
  if aError<>ECANVAS_SUCCESS then
    if aMsg='' then
      Raise Exception.CreateFmt('Canvas Operation failed %d',[aError])
    else
      Raise Exception.CreateFmt('%s : Error code %d',[aMsg,aError]);
end;
```

We can use this function to construct the other methods in the class:

```
constructor TWebCanvas.Create(aWidth, aHeight: Longint);
begin
  Check(__webcanvas_allocate(aWidth,aHeight,@FCanvasID), 'Failed to create web canvas');
  FWidth:=aWidth;
  FHeight:=aHeight;
end;

procedure TWebCanvas.moveto(X: Longint; Y: Longint);
begin
  Check(__webcanvas_moveto(FCanvasID,X,Y));
end;

procedure TWebCanvas.FillText(X: Longint; Y: Longint; S: UTF8String);
begin
  Check(__webcanvas_filltext(FCanvasID,X,Y, PByte(PAnsichar(S)),Length(S)));
end;
```

As you can see, this is not such difficult code.
Note how the string is passed to the imported `__web_canvas_filltext` function:
This resembles the way strings are passed to **C APIs:**
as a pointer to a null-terminated memory buffer, and a string length.
The class is now ready be used, and it can be used just as any class would be used in the browser itself:

```
Var
  aCanvas : TWebCanvas;
begin
  aCanvas:=TWebCanvas.Create(150,150);
  With aCanvas do
    try
      BeginPath;
      MoveTo(25,25);
      LineTo(125,125);
      FillText(8,8,'Greetings on the WebAssembly Canvas!');
    finally
      free;
    end;
end.
```

The complete code can be found in the demos of **Pas2JS.**
This concludes the **WebAssembly** side of things. So how do we go about creating the implementation in **Javascript**? We create a descendent of `TImportExtension` called `TWACanvas` - we present only the relevant calls here:

```
TWACanvas = class(TImportExtension)
Protected
  function GetCanvas(aID : TCanvasID) : TJSCanvasRenderingContext2D;
  function allocate(SizeX, SizeY : Longint; aID: Longint): TCanvasError;
  function moveto(aID : TCanvasID; X,Y : Longint): TCanvasError;
  function FillText(aID : TCanvasID; X,Y : Longint;
        aText : Longint; aTextLen : Longint ): TCanvasError;
Public
  Constructor Create(aEnv : TPas2JSWASIEnvironment); override;
  Procedure FillImportObject(aObject : TJSObject); override;
  Function ImportName : String; override;
  Property CanvasParent : TJSHTMLELement;
end;
```

The **`ImportName`** and **`FillImportObject`** methods must be overridden and this looks like this:

```
procedure TWACanvas.FillImportObject(aObject: TJSObject);
begin
  aObject['allocate']:=@allocate;
  aObject['moveto']:=@moveto;
  aObject['filltext']:=@FillText;
end;

function TWACanvas.ImportName: String;
begin
  Result:='web_canvas';
end;
```

You can see that the names used are the same names as used to import the functions. This is very important: if one of the names is missing, the **Javascript WebAssembly** runtime will raise a **`LinkError`** exception when instantiating the **WebAssembly** runtime.
The **`Allocate`** function is called to create a new canvas.

```
function TWACanvas.allocate(SizeX : Longint;
  SizeY : Longint;
  aID: Longint): TCanvasError;
Var
   C : TJSElement;
  V : TJSDataView;
  SID : String;
begin
  C:=window.document.createElement('CANVAS');
  CanvasParent.AppendChild(C);
  Inc(FCurrentID);
  SID:=IntToStr(FCurrentID);
  FCanvases[SID]:=TJSHTMLCanvasElement(c).getcontext('2d');
  V:=getModuleMemoryDataView;
  v.setUint32(aID, FCurrentID, env.IsLittleEndian);
  Result:=ECANVAS_SUCCESS;
end;
```

It creates a new **CANVAS** html element, and attaches it to the **HTML** Element specified in the **`CanvasParent`** property. It is then stored with a unique ID in a map with allocated canvas elements, so later on the canvas can be retrieved using this unique ID. This mechanism is arbitrary, in a real-world application, the canvas element to use would probably be communicated to the **WEBASSEMBLY** runtime.
The interesting thing here is how the **ID** is communciated to the **WebAssembly** runtime: the **WebAssembly** definition of the **`__webcanvas_allocate`** call uses a pointer to an address where the **ID** must be stored (*i.e. it is a var parameter*):

```
function __webcanvas_allocate(
SizeX : Longint;
SIzeY : Longint;
aID: PCanvasID
): TCanvasError;
```

The pointer is converted to an integer (*the index in memory*). The memory of the **webassembly runtime** is exposed to the **Javascript** environment. The `getModuleMemoryDataView` call returns the memory as a `TJSDataView` class (a standard **Javascript** class): in essence an object that can be used to read and write to an underlying array.

This is then also how the **ID** is communicated to the **webassembly runtime**, it is written directly to the WebAssembly memory using the `setUint32` method of `TJSDataView`.
The `MoveTo` function is actually quite easy. It gets 3 integers as parameters, and does not need memory access. It starts by mapping the canvas **ID** to an actual canvas renderingcontext,

```
function TWACanvas.moveto(aID : TCanvasID; X : Longint;Y : Longint): TCanvasError;
Var
C : TJSCanvasRenderingContext2D;
begin
Result:=ECANVAS_NOCANVAS;
C:=GetCanvas(aID);
if Assigned(C) then
begin
C.moveto(X,Y);
Result:=ECANVAS_SUCCESS;
 end;
end;
```

The **WebAssembly** program uses a **UTF8**-encoded ansistring to communicate a string. The `TPas2JSWASIEnvironment` class has a convenience function that reads an **UTF8** string from the **WebAssembly** memory, given a location and length: `GetUTF8StringFromMem`. This function is used here to retrieve the string to be written on the canvas.
To use this class and have it imported in the **WebAssembly** runtime, we just need to create it after we have created the environment:

```
FWasiEnv:=TPas2JSWASIEnvironment.Create;
FWasiEnv.OnStdErrorWrite:=@DoWrite;
FWasiEnv.OnStdOutputWrite:=@DoWrite;
FWACanvas:=TWACanvas.Create(FWasiEnv);
FWACanvas.CanvasParent:=GetHTMLElement('canvases');
```

That's all there is to it. To remove the extension, it is sufficient to destroy it.
The result of the test program can be seen on
`https://www.freepascal.org/~michael/pas2js-demos/wasienv/canvas/`
it will look like figure 9 on page 17.

7 CONCLUSION

In this article, we've shown how **Free Pascal** can be used to write **WebAssembly** programs. We also demonstrated how **Pas2JS** can be used to host the **WebAssembly** program in a browser, and how to extend the **WebAssembly** environment with custom functions.

The compiler support for webassembly is quite stable, but support for the Browser hosting using **PAS2JS** is quite new (*for example standard file support needs still to be added to it*), and will surely need some time to mature: we'll report about the progress in future contributions.

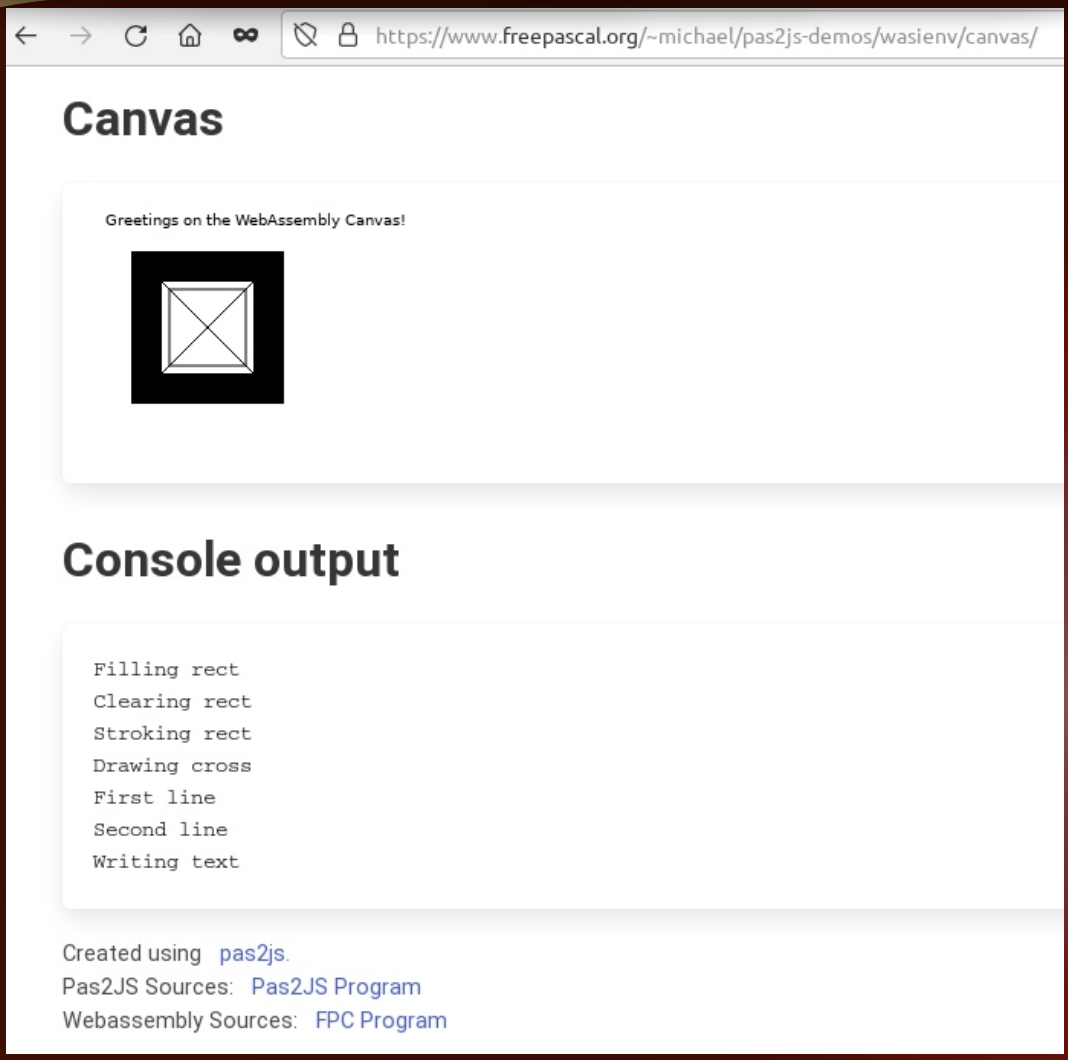


Figure 9: The canvas demo

COMPONENTS 4 DEVELOPERS

kbmFMX Std/Pro v. 1.50.00 released
JAN 1, 2022 KIMBOMADSEN
We are happy to announce an update to
kbmFMX Standard
and Professional Edition.
kbmFMX Standard Edition
is bundled with kbmMemTable...

COMPONENTS 4 DEVELOPERS

KBMMW PROFESSIONAL AND ENTERPRISE EDITION V. 5.18.00 RELEASED!

- **RAD Studio XE5 to 11 Alexandria supported**
- Win32, Win64, Linux64, Android, IOS 32, IOS 64 and OSX client and server support
- Native high performance 100% developer defined application server
- Full support for centralized and distributed load balancing and failover
- Advanced ORM/OPF support including support of existing databases
- Advanced logging support
- Advanced configuration framework
- Advanced scheduling support for easy access to multithread programming
- Advanced smart service and clients for very easy publication of functionality
- High quality random functions.
- High quality pronouncable password generators.
- High performance LZ4 and Jpeg compression
- Complete object notation framework including full support for YAML, BSON, Messagepack, JSON and XML
- Advanced object and value marshalling to and from YAML, BSON, Messagepack, JSON and XML
- High performance native TCP transport support
- High performance HTTPSys transport for Windows.
- CORS support in REST/HTML services.
- Native PHP, Java, OCX, ANSI C, C#, Apache Flex client support!

kbmMemTable is the fastest and most feature rich in memory table for Embarcadero products.

- **Easily supports large datasets with millions of records**
- **Easy data streaming support**
- **Optional to use native SQL engine**
- **Supports nested transactions and undo**
- **Native and fast build in M/D, aggregation/grouping, range selection features**
- **Advanced indexing features for extreme performance**

- New I18N context sensitive internationalization framework to make your applications multilingual.
- New ORM LINQ support for Delete and Update.
- Comments support in YAML.
- New StreamSec TLS v4 support (by StreamSec)
- Many other feature improvements and fixes.

Please visit
http://www.components4developers.com
for more information about kbmMW

- High speed, unified database access (35+ supported database APIs) with connection pooling, metadata and data caching on all tiers
- Multi head access to the application server, via REST/AJAX, native binary, Publish/Subscribe, SOAP, XML, RTMP from web browsers, embedded devices, linked application servers, PCs, mobile devices, Java systems and many more clients
- Complete support for hosting FastCGI based applications (PHP/Ruby/Perl/Python typically)
- Native complete AMQP 0.91 support (Advanced Message Queuing Protocol)
- Complete end 2 end secure brandable Remote Desktop with near realtime HD video, 8 monitor support, texture detection, compression and clipboard sharing.
- Bundling kbmMemTable Professional which is the fastest and most feature rich in memory table for Embarcadero products.

COMPONENTS 4 DEVELOPERS

EESB, SOA,MoM, EAI TOOLS FOR INTELLIGENT SOLUTIONS. kbmMW IS THE PREMIERE N-TIER PRODUCT FOR DELPHI / C++BUILDER