Pharo CIG, Round Two: Beyond the Bridge

Growing a native library ecosystem for Pharo

Esteban Lorenzano · ESUG 2026 · Plovdiv

2024 → 2026

01 · The callback

Two years ago, I talked about building a bridge.

The problem: Pharo had a capable FFI, yet using a native library still meant writing every binding by hand.

The FFI was not the real obstacle.

Pharo already had

Portable calls.
Callbacks.
Thread support.
A mature UnifiedFFI.

≠

What projects still needed

Hundreds of functions, structs, enums, types, comments and platform details translated by hand.

The bottleneck was the binding work.

One library is often a small software project.

Copy
Turn headers into classes and FFI methods.

Debug
Get calling conventions, types, structs and ownership right.

Maintain
Repeat when the library, Pharo, or a platform changes.

A powerful FFI is not enough if every project must rebuild the same door into the native world.

CIG turns a C header into a repeatable process.

1. Header

Use the library's existing C declaration, **or make your own**.

→

2. libclang

Parse what the header actually means.

→

3. Pharo

Generate low-level bindings from a recipe.

Not "magic" – a reproducible translation.

The recipe is the important artifact.

```
"This is a very simple example"

CigCLibraryGenerator new
  prefix: 'QR';
  packageName: 'LibQREncode';
  libraryName: 'QREncode';
  unixLibraryName: 'libqrencode.so';
  import: '/home/esteban/dev/vm/libqrencode/qrencode.h';
  cTypedef: '_QRinput' as: #void;
  cTypedef: '_QRinput_Struct' as: #void;
  generate.
```

It records

- what we import
- where headers live
- naming decisions
- required configuration
- how to generate it again

A binding can be rebuilt, not rediscovered.

```
"... and this is a complex one 😉"

CigCLibraryGenerator new
	prefix: 'SSL';
	libraryName: 'ssl';
	cExcludePath: '/usr/include';
	cIncludePath: '/usr/include/openssl';
	importUnit: '
		#ifdef CIG_INCLUDE_TIME
		/* taken from time.h, to not import the whole header */
		struct timeval
		{
		  long tv_sec;		/* Seconds.  */
		  long tv_usec;		/* Microseconds.  */
		};
		struct tm
		{
		  int tm_sec;		/* Seconds.	[0-60] (1 leap second) */
		  int tm_min;		/* Minutes.	[0-59] */
		  int tm_hour;		/* Hours.	[0-23] */
		  int tm_mday;		/* Day.		[1-31] */
		  int tm_mon;		/* Month.	[0-11] */
		  int tm_year;		/* Year	- 1900.  */
		  int tm_wday;		/* Day of week.	[0-6] */
		  int tm_yday;		/* Days in year.[0-365]	*/
		  int tm_isdst;		/* DST.		[-1/0/1]*/
		  long int tm_gmtoff;	/* Seconds east of UTC.  */
		  const char *tm_zone;	/* Timezone abbreviation.  */
		};
		#endif
		#import <openssl/aes.h>
		#import <openssl/asn1.h>
		#import <openssl/asn1t.h>
		#import <openssl/bio.h>
		#import <openssl/blowfish.h>
		#import <openssl/bn.h>
		#import <openssl/buffer.h>
		#import <openssl/camellia.h>
		#import <openssl/cast.h>
		#import <openssl/cmac.h>
		#import <openssl/conf.h>
		#import <openssl/core.h>
		#import <openssl/core_names.h>
		#import <openssl/crypto.h>
		#import <openssl/ct.h>
		#import <openssl/des.h>
		#import <openssl/dh.h>
		#import <openssl/dsa.h>
		#import <openssl/ec.h>
```

```
'
		#import <openssl/ecdh.h>
		#import <openssl/ecdsa.h>
		#import <openssl/engine.h>
		#import <openssl/err.h>
		#import <openssl/evp.h>
		#import <openssl/hmac.h>
		#import <openssl/idea.h>
		#import <openssl/kdf.h>
		#import <openssl/md5.h>
		#import <openssl/modes.h>
		#import <openssl/objects.h>
		#import <openssl/ocsp.h>
		#import <openssl/opensslconf.h>
		#import <openssl/opensslv.h>
		#import <openssl/pem.h>
		#import <openssl/pkcs12.h>
		#import <openssl/pkcs7.h>
		#import <openssl/provider.h>
		#import <openssl/rand.h>
		#import <openssl/rc2.h>
		#import <openssl/rc4.h>
		#import <openssl/ripemd.h>
		#import <openssl/rsa.h>
		#import <openssl/sha.h>
		#import <openssl/srtp.h>
		#import <openssl/ssl.h>
		#import <openssl/tls1.h>
		#import <openssl/ts.h>
		#import <openssl/ui.h>
		#import <openssl/whrlpool.h>
		#import <openssl/x509.h>
		#import <openssl/x509v3.h>';
	cTypedef: 'pthread_once_t' as: #void;
	cTypedef: 'pthread_key_t' as: #void;
	cTypedef: 'pthread_t' as: #void;
	cTypedef: 'intmax_t' as: #long;
	cTypedef: 'uintmax_t' as: #ulong;
	cTypedef: 'lhash_st_CONF_VALUE' as: 'lhash_st';
	cTypedef: 'pkcs7_st' as: 'void';
	cDefine: 'CIG_INCLUDE_TIME';
	in: [ :this |
		#(
		'ACCESS_DESCRIPTION' 'ASIdOrRange' 'ASN1_INTEGER' 'ASN1_OBJECT' 'ASN1_TYPE' 'ASN1_STRING'
		'ASN1_UTF8STRING' 'CONF_VALUE' 'CONF_VALUE' 'DIST_POINT' 'GENERAL_NAME' 'GENERAL_SUBTREE'
		'IPAddressFamily' 'IPAddressOrRange' 'OPENSSL_CSTRING' 'OPENSSL_STRING'
		'OSSL_ALLOWED_ATTRIBUTES_CHOICE' 'OSSL_ALLOWED_ATTRIBUTES_ITEM' 'OSSL_ATTRIBUTE_MAPPING'
		'OSSL_DAY_TIME_BAND' 'OSSL_ROLE_SPEC_CERT_ID' 'OSSL_TIME_PERIOD' 'PKCS12_SAFEBAG' 'PKCS7'
		'PKCS7_RECIP_INFO' 'PKCS7_SIGNER_INFO' 'POLICYINFO' 'POLICYQUALINFO' 'POLICY_MAPPING'
		'PROFESSION_INFO' 'SCT' 'SRTP_PROTECTION_PROFILE' 'SSL_CIPHER' 'SSL_COMP' 'SSL_SESSION'
		'SXNETID' 'USERNOTICE' 'X509' 'X509_ALGOR' 'X509_ATTRIBUTE' 'X509_CRL' 'X509_EXTENSION'
		'X509_INFO' 'X509_NAME' 'X509_NAME_ENTRY' 'X509_OBJECT' 'X509_POLICY_NODE' 'X509_REVOKED'
		'void' 'ADMISSIONS')
		do: [ :each | this cTypedef: ('stack_st_', each) asSymbol as: 'stack_st' ] ];
	withConstantGenerator: [ :gen | gen groupByHeader ];
	generate.
```

Advanced customisation

Element transformation

Return pointer instead string, change signature, make structure opaque, **anything required**.

Name tweaking

Strip/modify prefixes, make camelCase, rename elements, group protocols by header...

Constant generation

Get from specific headers, group constants by different strategies (header, prefix, no group).

Macro generation (TODO)

We can generate helper libraries to expose macros as functions. **Do we want that?**

Make your own!

Generation gives us the low-level layer quickly.

Generated

Functions, structs, enums, constants, library lookup, FFI signatures.

Built on top

Readable APIs, ownership rules, conveniences, Pharo objects, examples.

Generated bindings are not the end of API design.
They make API design start from a usable base.

02 · What changed

The bridge is now usable.

Not complete. Not frictionless. But useful enough that it is starting to disappear into real work.

CIG is already being used as infrastructure.

Pharo 11 — Generated libraries remain usable through UnifiedFFI reloading changes.

Pulsar — CIG is used extensively for native integrations.

GTK itself remains special: it needs a dedicated generator.

SDL3 — Pharo-on-SDL3 work is another place where generated native access becomes strategically useful.

The tool starts to matter when other projects depend on it.

But private bindings do not make an ecosystem.

Without sharing

Every project has its own wrapper, fixes its own edge cases, and loses its work when the project stops.

→

With a common place

A library binding becomes a starting point for other projects, not a private implementation detail.

03 · Beyond the bridge

We need a place to arrive: the Pharo Interface Garden.

A shared space for native libraries that Pharo projects can use, improve and grow together.

A garden is more than a package catalogue.

Seeds

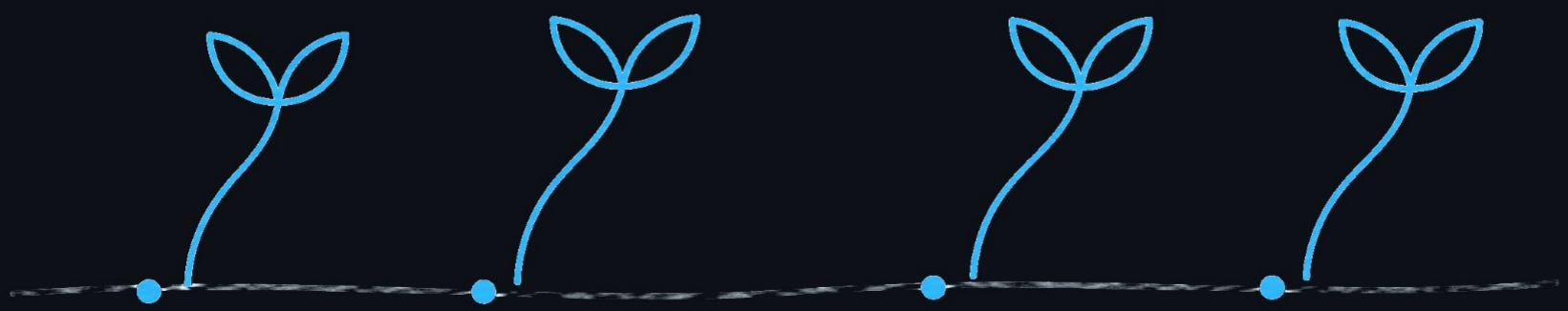
Recipes, headers, build notes, platform assumptions.

Plants

Generated low-level bindings with compatibility work.

Paths

Examples, documentation, tests and Pharo-level wrappers.



What can live in the Garden?

Graphics & UI
Rendering, fonts, SVG, multimedia and platform toolkits.

Developer building blocks
Parsers, engines, compression, data processing and useful C libraries.

System access
Operating-system APIs, file formats, networking and hardware support.

Higher-level packages
Idiomatic Pharo APIs built over stable generated foundations.

The Garden should make a library easier to discover, reuse and maintain.

Contribution should be a normal development path.

I need a library

Start from a concrete Pharo project.

→

I write a recipe

Generate and validate the low-level binding.

→

We share it

Publish, document, test and improve it together.

Need-driven work becomes community infrastructure.

Generated code gives us a common floor.
Pharo design builds the house.

Floor: generated binding

Close to the C API. Broad coverage. Easy to regenerate and compare with the header.

House: Pharo library

Objects, lifetime management, errors, collections, clean naming and useful examples.

The generator is not trying to automate the decisions that require Pharo judgement.

04 · The frontier

C is a wide territory.
C++ is still a frontier.

And it is hard for real technical reasons — not because more parsing effort was missing.

Why C++ needs a different answer.

ABI
No single universal binary contract like C.

Language
Templates, namespaces, references, standard-library types.

Ownership
Lifetime, exceptions, object models and wrappers need explicit choices.

CIG can help explore and generate pieces of C++, but it does not yet make arbitrary C++ libraries routine.

Why not to use an LLM to do the work?

After all, this is supposedly where LLMs shine.

Non deterministic

After several tests. Conclusion is: LLMs are currently incapable of repeat what CIG does.

→

Deterministic

They can integrate CIG and the projects already in the garden and with that info generate a good **recipe**.

The promise is not: “everything is automatic.”

What CIG should make cheap

- ordinary C headers
- repetition and boilerplate
- regeneration after library changes
- shared low-level foundations

Where people add value

- ownership and safety
- Pharo abstractions
- complex library adaptation

The next bottleneck is not calling C. It is making native access shared infrastructure.

FFI gives us capability.
CIG gives us repeatability.
The Garden can give us an ecosystem.

The bridge exists.

Let's build the place on the other side.

https://github.com/pharo-cig

Bring a library · write a recipe · grow the Garden