Inner Workings of Functions

Below, we go more into depth of what is happening under the hood when you create a function in an Aztec contract and what the attributes are really doing.

To get a more practical understanding about functions, readthe rest of this section .

Private functions

Aztec.nr uses an attribute system to annotate a function's type. Annotating a function with the#[aztec(private)] attribute tells the framework that this is a private function that will be executed on a users device. The compiler will create a circuit to define this function.

aztec(private)

is just syntactic sugar. At compile time, the Aztec.nr framework inserts code that allows the function to interact with thekernel .

To help illustrate how this interacts with the internals of Aztec and its kernel circuits, we can take an example private function, and explore what it looks like after Aztec.nr's macro expansion.

Before expansion

simple_macro_example

[aztec(private)]

fn

simple_macro_example (a :

Field , b :

Field)

->

Field

{ a + b }Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L254-L259

After expansion

simple_macro_example_expanded fn

simple_macro_example_expanded (// ************ // The private context inputs are made available to the circuit by the kernel inputs :

PrivateContextInputs , // ************

// Our original inputs! a :

Field , b :

Field

// The actual return type of our circuit is the PrivateCircuitPublicInputs struct, this will be the // input to our kernel!)

-> distinct pub

PrivateCircuitPublicInputs

{ // ************ // The hasher is a structure used to generate a hash of the circuits inputs. let

mut hasher =

Hasher :: new ( ) ; hasher . add (a) ; hasher . add (b) ;

// The context object is created with the inputs and the hash of the inputs let

mut context =

PrivateContext :: new (inputs , hasher . hash ( )) ;

let

mut storage =

Storage :: init (Context :: private (& mut context)) ; //************

// Our actual program let result = a + b ;

// ************ // Return values are pushed into the context context . return_values . push (result) ;

// The context is returned to be consumed by the kernel circuit! context . finish ( ) //************ } Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L261-L310

The expansion broken down?

Viewing the expanded Aztec contract uncovers a lot about how Aztec contracts interact with the kernel . To aid with developing intuition, we will break down each inserted line.

Receiving context from the kernel.

context-example-inputs inputs :

PrivateContextInputs , Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L265-L267 Private function calls are able to interact with each other through orchestration from within the kernel circuit . The kernel circuit forwards information to each contract function (recall each contract function is a circuit). This information then becomes part of the private context. For example, within each private function we can access some global variables. To access them we can call on thecontext , e.g.context.chain_id() . The value of the chain ID comes from the values passed into the circuit from the kernel.

The kernel checks that all of the values passed to each circuit in a function call are the same.

Returning the context to the kernel.

context-example-return)

-> distinct pub

PrivateCircuitPublicInputs

{ Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L274-L276 The contract function must return information about the execution back to the kernel. This is done through a rigid structure we call thePrivateCircuitPublicInputs .

Why is it called thePrivateCircuitPublicInputs ? When verifying zk programs, return values are not computed at verification runtime, rather expected return values are provided as inputs and checked for correctness. Hence, the return values are considered public inputs. This structure contains a host of information about the executed program. It will contain any newly created nullifiers, any messages to be sent to l2 and most importantly it will contain the return values of the function.

Hashing the function inputs.

context-example-hasher let

mut hasher =

Hasher :: new ( ) ; hasher . add (a) ; hasher . add (b) ; Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L279-L283 What is the hasher and why is it needed?

Inside the kernel circuits, the inputs to functions are reduced to a single value; the inputs hash. This prevents the need for multiple different kernel circuits; each supporting differing numbers of inputs. The hasher abstraction that allows us to create an array of all of the inputs that can be reduced to a single value.

Creating the function's context.

context-example-context let

mut context =

PrivateContext :: new (inputs , hasher . hash ( )) ;Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L286-L288 Each Aztec function has access to acontext object. This object, although labelled a global variable, is created locally on a users' device. It is initialized from the inputs provided by the kernel, and a hash of the function's inputs.

context-example-context-return context . return_values . push (result) ;Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L300-L302 We use the kernel to pass information between circuits. This means that the return values of functions must also be passed to the kernel (where they can be later passed on to another function). We achieve this by pushing return values to the execution context, which we then pass to the kernel.

Making the contract's storage available

storage-example-context let

mut storage =

Storage :: init (Context :: private (& mut context)) ;Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L290-L292 When aStorage struct is declared within a contract, thestorage keyword is made available. As shown in the macro expansion above, this calls the init function on the storage struct with the current function's context.

Any state variables declared in theStorage struct can now be accessed as normal struct members.

Returning the function context to the kernel.

context-example-finish context . finish ( )Source code: noir-projects/noir-contracts/contracts/docs_example_contract/src/main.nr#L305-L307 This function takes the application context, and converts it into thePrivateCircuitPublicInputs structure. This structure is then passed to the kernel circuit.

Unconstrained functions

Defining a function asunconstrained tells Aztec to simulate it completely client-side in theACIR simulator without generating proofs. They are useful for extracting information from a user through anoracle .

When an unconstrained function is called, it prompts the ACIR simulator to

1. generate the execution environment
2. execute the function within this environment

To generate the environment, the simulator gets the blockheader from thePXE database and passes it along with the contract address toViewDataOracle . This creates a context that simulates the state of the blockchain at a specific block, allowing the unconstrained function to access and interact with blockchain data as it would appear in that block, but without affecting the actual blockchain state.

Once the execution environment is created,execute_unconstrained_function is invoked:

execute_unconstrained_function /* *Execute an unconstrained function and return the decoded values./* export

async

function

executeUnconstrainedFunction (oracle : ViewDataOracle , artifact : FunctionArtifactWithDebugMetadata , contractAddress : AztecAddress , functionData : FunctionData , args : Fr [ ] , log =

createDebugLogger ('aztec:simulator:unconstrained_execution') ,) :

Promise < DecodedReturn

{ const functionSelector = functionData . selector ; log (Executing unconstrained function { contractAddress } : { functionSelector }) ;

const acir = Buffer . from (artifact . bytecode ,

'base64') ; const initialWitness =

toACVMWitness (0 , args) ; const

{ partialWitness }

=

await

acvm (await AcirSimulator . getSolver ( ) , acir , initialWitness , new

Oracle (oracle) ,) . catch ((err : Error)

=>

{ throw

new

ExecutionError (err . message , { contractAddress , functionSelector , } , extractCallStack (err , artifact . debug) , { cause : err } ,) ; }) ;

return

decodeReturnValues (artifact ,

extractReturnWitness (acir , partialWitness)) ; }Source code: yarn-project/simulator/src/client/unconstrained_execution.ts#L13-L49 This:

1. Prepares the ACIR for execution
2. Convertsargs
3. into a format suitable for the ACVM (Abstract Circuit Virtual Machine), creating an initial witness (witness = set of inputs required to compute the function).args
4. might be an oracle to request a user's balance
5. Executes the function in the ACVM, which involves running the ACIR with the initial witness and the context. If requesting a user's balance, this would query the balance from the PXE database
6. Extracts the return values from thepartialWitness
7. and decodes them based on the artifact to get the final function output. Theartifact
8. is the compiled output of the contract, and has information like the function signature, parameter types, and return types Edit this page

Previous How to define compute_note_hash_and_nullifier Next Storage