The Quiche Language Specification

# Introduction

Quiche is a language derived from Pascal and intended to target 8-bit microprocessors. As such it includes enhancements to allow it to generate more efficient code on such processors. It also includes syntax shortcuts to simplify coding and allow more compact programs to be written, but without compromising the traditional strengths and pleasures of Pascal.

# Simple Types

The following ‘simple types’ are available:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type name | Suffix | Range | Byte size | Type | Notes |
| Byte | ## | 0..255 | 1 | Unsigned 8-bit |  |
| Word | # | 0..65535 | 2 | Unsigned 16-bit |  |
| Int8 | %% | -128..127 | 1 | Signed 8-bit | Internal use only[1] |
| Int16 |  | -32768..32767 | 2 | Signed 16-bit |  |
| Integer | % | -32768..32767 | 2 | Signed 16-bit | Synonym for Int16[2] |
| Real | ! | [3] | [3] | [3] | [3] |
| Char | $$ | Chr(0)..Chr(255) | 1 | ASCII character |  |
| String | $ |  | Variable | ‘Pascal’ string | [3] |
| Boolean | ? | False..True | 1 | Boolean |  |
| Pointer | ^ | 0..65535 | 2 | Pointer | [4] |

[1] – Available for possible future use. Currently only used internally for compiler optimisations.

[2] – Integer is platform specific, Int16 is platform independent. The size of an Integer may change if Quiche is ever available on 32-bit or 64-bit platforms.

[3] – Not currently implemented but planned for future use.

[4] – Essentially the same as a Word but using the appropriate type can help the compiler with optimisations and code generation.

# 4. Constant Literals

Hexadecimal constants begin with a $ character, e.g.

a := $c000

Binary constants begin with a % character, e.g.

a := %11000111

Underscores within numeric constants are allowed but ignored. These can be used to break up the digits into blocks as desired:

a := 1\_000

b := $10\_ff

c := %00\_111\_000

Strings (not yet included) and characters are bracketed in single quotes:

S := ‘Hello world’

Single characters can also be specified with a leading hash symbol:

C := #32

Characters specified in this way can also use hexadecimal or binary,

D := #$20

E := #%00\_010\_000

Boolean values can be specified with the False and True keywords.

B := False

# Variable Declarations

Variables are declared with the var keyword and optionally initialised at the same time.

The standard Pascal syntax can be used:

var a: Integer

A variable can also be initialised at the time it is declared. Note how the syntax here mirrors that of a typed const declaration:

var b: Integer = 1

Quiche also allows a BASIC style type suffix symbol, here defining c as an integer. You can read more about type suffix symbols in the section on Types:

var c%

A type suffix is *only* allowed in a variable declaration. All other references, assignments, etc must be made without the suffix. The name of the variable is the identifier without the suffix, therefore it’s not possible to declare variables with the same name but different types. The suffix is merely a shorthand for convenience in declarations.

This type suffix form can also be initialised:

var d% := 2

## Type Inference

Quiche can use *type inference* to establish the type of the variable based on the type of the assignment expression:

var e := 3

Declarations are allowed ‘inline’ with code, the same as in the C language. Inline declarations are local to the block in which they are declared.

When the expression is a constant the following rules are used to determine the type of the variable being created.

|  |  |
| --- | --- |
| Expression type | Variable type |
| Floating point | Real[1] |
| Decimal integer | Integer |
| One or two digit hexadecimal[2] | Byte |
| Three or more digit hexadecimal[2] | Word |
| One to eight digit binary[2] | Byte |
| Nine or more digit binary[2] | Word |
| One character string | Char |
| Zero length string | String[1] |
| Two or more character string | String[1] |
| Boolean | Boolean |

[1] – Not yet implemented, for future use.

[2] – Leading zeros are included in the digit count, underscores are not.

Where the expression is a variable reference the created type will be the same as the variable referred to.

Where the expression is more complex, i.e. involving operators, the types will be extended as necessary. The exact nature of the extension depends on the operator and the types of the two values. Generally if either value is a Real the result with be a real. If either value is an Integer the result will be an Integer.

# Expressions and Operators

## Expression and Efficient Code

One of the stated goals for Quiche is the ability for it to generate efficient code on 8-bit processors. One of the key parts of generating efficient code is the way expressions are processed.

We have an expression which adds two Bytes (8-bit unsigned type) and assigns the result to a Word (16-bit unsigned type):

w := b1 + b2

Should the addition be a fast 8-bit operation or a slower 16-bit operation? Maybe we argue that the result is being assigned to a 16-bit variable so a 16-bit addition would be sensible. Now what happens with more complex expressions:

w := (b1 + b2) \* b3 – b4

Should the entire expression be computed using 16-bit values? Should it be computed using 8-bit operations and expanded to 16-bit at some point mid-expression? Should the entire expression be computed using 8-bit operations and expanded to a 16-bit when the final result is assigned to W?

Returning to the goals of the language, the aim is to enable the programmer to generate efficient code. Efficient code in this context entails using 8-bit operations where possible and expanding to 16-bit values only when required. Only the programmer knows what types are required for the expression to not overflow. Therefore, the decision as to whether to use 8-bit operations, or to expand values to 16-bit operations should be left in the hands of the programmer.

If a programmer wants to change the native type of an operation he can do so using a typecast. So in the previous example he would write:

w := (Word(b1) + b2) \* b3 – b4

Raising the type of b1 to a Word would raise the type of the addition to Word. This would raise the type of all the following operations to Word. To keep the addition as an 8-bit operation but use 16-bits for the multiplication and subtraction the coder can write:

w := Word(b1 + b2) \* b3 – b4

If the default situation where for the expression to adopt the type of the assignment it would be necessary for the programmer to specify the types of all the operations. I’m not aware of any language with a syntax which does allow that, or how one would be created which wasn’t overly complex.

In summary, the type of an operation, and therefore the type of the result of each operation is determined based on the types of the operands.

## Mixing Operand Types

We can now move on to discussing operations where the operators are of mixed types.

As we saw in the earlier example, if both operands are unsigned and of different sizes the operation Type can be determined by the Type of the larger operand (i.e the 16-bit one) with the smaller operand (the 8-bit one) be extended to 16-bits.

For a mix of signed operands of different sizes the same rule can be applied with the 8-bit value being extended to 16-bit.

Things start to become more complex when we combine signed and unsigned operands. If we add an unsigned value to a signed value should the result be signed or unsigned? Bear in mind here that signed and unsigned values each have a different range of possible values. In binary terms that means that the same raw value (e.g. in a register) can represent different values depending on the Type we assume the value to be. A 16-bit integer value between zero and 32767 has the same value in both signed and unsigned systems, but the values 32768 to 65535 of an unsigned value represent the values -32768 to -1 if viewed as a signed number.

It's worth noting that this issue only applies if we are checking the result for *overflow* (i.e. whether the result is a valid result). There are valid reasons why an overflowing result may be a valid one but, in general, overflow checking is a necessary part of checking that our code hasn’t malfunctioned on some way.

Looking at a concrete example, where S is a signed variable and U is an unsigned variable we can have the expression:

r := s + u

One interpretation says that we are adding an unsigned value to a signed one, say the amount of a deposit to a bank balance and require a signed result.

The other interpretation says we’re adding a signed value to an unsigned one, such as stepping forwards or backwards through an array index. In this case we expect an unsigned result.

How should the compiler interpret such a scenario?

Quiche takes the view that, to most programmers, using Integer is the default way to code for ‘general’ numbers. Therefore the unsigned value should dominate. In other words a combination of signed and unsigned values should result in a signed result.

That’s cool, but much coding on 8-bit machines involves addresses. And addresses are. Of course, unsigned. When we add an offset to a memory address we expect to get a memory address back. An unsigned memory address.

Some though was put into whether this could be accomplished via the order of the operands. Ie. if the first operand is unsigned then the result would be unsigned, if the first operand was signed then the result would be signed. It was felt, however, that this would be both confusing and potential trap for new or occasional users of the language, and there’s nothing worse for a new user than spending ages debugging a weird compiler quirk.

The solution is actually a rather obvious one. The language includes the Pointer type to refer to memory addresses. Quiche therefore assumes that any operation involving Pointers is expecting an unsigned result in a Pointer type variable. Thus if the programmer wants an expression which mixes signed and unsigned types to yield an unsigned result they should use a Pointer type.

## Type Precedence

Thus Quiche has a concept called *type precedence*, which specifies how combinations of Types flow through an expression. In other words, when an operator is given operands of more than one type, how is the type of the operand determined, and how is the type of the result determined.

Note that these rules only apply to numeric types, and they also depend on the specific operator. I.e. some operators (e.g. addition) expect two operands of the same type whereas for other operations the types of the operands are independent (e.g. SHL – left shift). Exact details for each operators will be clarified later in this chapter.

The rules are as follows. They are worked through from top to bottom until an answer is given:

* If either operand is a Real (floating point) then the operation and result will be Real.
* If either operand is a Pointer and the other operand is unsigned the operation and result will be unsigned 16-bit (Pointer).
* If either operand is a Pointer and the other is signed the operation will be *mixed* and the result will be 16-bit unsigned (Pointer).
* If both operands are Byte type (unsigned 8-bit) the operation and result will be unsigned 8-bit.
* If both operands are unsigned (Word or Byte) the operand and result will be unsigned 16-bit (Word).
* If both operations are signed 8-bit the operation will be signed 8-bit (if the operation supports 8-bits, otherwise the 8-bit signed will be treated as 16-bit signed).
* If Both operands are signed the operation and result will be 16-bit unsigned.
* If one operand is signed and the other is 8-bit unsigned (Byte) the operation and result will be 16-bit signed.
* If one operand is signed and the other is 16-bit unsigned (Word) the operation will be treated as *mixed* and the result as 16-bit signed.

A *mixed* operation will involve the compiler generating slightly different code, or using a different library routine, to enable the correct result to be calculated. This is only really relevant if you’re reading or modifying the source code or the operation or primitives tables.

## Operators

This brings us neatly to details of the operators which are available in Quiche.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operation | Symbol | Operator precedence | Uses type precedence | Int8 available | Notes |
| Addition | + |  | Yes | Yes |  |
| Subtraction | - |  | Yes | Yes |  |
| Multiplication | \* |  | Yes | No |  |
| Real division | / |  | No | No | Floating point operation and result[1] |
| Integer division | div |  | Yes[2] | No | Integer operation and result |
| Modulus | mod |  | Yes | No | Integer only. Dividend must be positive |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

[1] Reals are not yet available – for future use.

[2] Excluding Reals which need to use ‘/’ operator

# Efficiency Tips

* For numeric types use 8-bit values and variables where possible, i.e. Byte and Int8 types. (but bear in mind that Int8 support is very limited and they will often be converted to Int16 values).
* Use unsigned numeric types in preference to signed ones. When adding unsigned integers on the Z80 we can use the Carry flag to test for overflow. We can do that with the ADD HL,rp instruction which is a single byte opcode. To test a signed addition for overflow we need to use the Parity/overflow flag. Sadly the ADD HL,rp instruction does not affect the parity/overflow flag. Instead we need to use the ADC HL,rp instruction. Not only is this one of the two byte extended opcodes, but we also need to take steps to clear the Carry flag before the operation. We have, thus, gone from a single byte operation to a three byte operation. Also, testing the Carry flag can be done with the a JR instruction (if the jump is small enough) whereas the parity/overflow flag requires a JP (or CALL).