The Kingfisher

Programming Language

Malcolm North, December 2024

Table of Contents

1. Foreword 1
2. Abstract 3
I: Getting Started 5
    3. The Kingfisher Language 7
        Design Philosophy 7
        System Architecture 7
    4. Conventions and Standards 9
        Naming Standards 9
        Documentation Standards 9
    5. The Interactive Development Environment 11
    6. Using the REPL 13
        The Command Line Interface (CLI) 13
        Comments in Kingfisher 13
    7. Stack-Based Programming Fundamentals 15
        Understanding Postfix Notation 15
    8. Number System 17
    9. Error Management 19
        Error Types and Handling 19
    10. Stack Signatures 21
        Type Categories 21
    11. The Parameter Stack 23
        Stack Signatures 23
        Stack Operators 23
    12. Arithmetic Operators 25
        Bitwise Operators 25
        General Operators 26
    13. Built-in Types 29
        Scalar Types 29
    14. Constants 31
        Declaring Constants 31
    15. Variables 33
        Variable Construction 33
        Variable Operations 34
    16. Collections 37
    17. Arrays 39
        Array Slices 39
    18. Strings 43
        String Slices 43

19. StrArrays . . . 45
StrArrSlices . . . 45
20. Definitions . . . 47
II: Language Fundamentals . . . 51
21. Program Organisation . . . 53
Bootstrap . . . 53
Vocabularies and Chains . . . 54
Modules . . . 56
Aliases . . . 56
Scope and Lifetime Rules . . . 57
Error Handling . . . 58
22. Type Definitions and Linked Methods . . . 59
Type Definitions . . . 59
Linked Methods . . . 61
Type Field Vocabulary . . . 61
Datasets . . . 64
23. Control Flow . . . 67
24. Boolean Operations . . . 69
Conditional Operations . . . 71
Error Handling . . . 74
Branching . . . 75
Iterators and Range . . . 78
III: Assembly Language and System Fundamentals . . . 83
25. Assembly Language . . . 85
Design Philosophy . . . 85
Assembly Language Core . . . 86
Instruction Set Support . . . 87
Assembly Directives . . . 90
Assembler . . . 91
Macro Processing . . . 92
26. System Data Structures . . . 95
27. Interrupts . . . 97
Interrupt Handling System . . . 97
IV: Runtime Features . . . 101
28. Introduction . . . 103
29. I/O Subsystem . . . 105
Console I/O . . . 105
Stream Definitions . . . 105
File I/O . . . 108
Operating System Management . . . 113
30. System Primitives . . . 115

31. Memory Management . . . 117
Dictionary Operations . . . 117
Heap Operations . . . 117
V: Kingfisher Development . . . 119
32. The Software Development Lifecycle (SDLC) . . . 121
Development Environment . . . 121
Talon IDE . . . 122
Example Application . . . 123
Another Example Application . . . 126
VI: Reference Sections . . . 133
33. Historical Context . . . 135
34. System References . . . 137
Error Message Reference . . . 137
35. Technical Background . . . 139
System Architecture . . . 139
Development References . . . 142
Perch CLI Reference . . . 143
Talon IDE Reference . . . 144
36. Fundamental System Structures . . . 149
Dictionary Entries . . . 149
Type System . . . 150
The Dictionary . . . 151
Dictionary Concepts . . . 151
Dictionary Reference . . . 152
Dictionary Management . . . 153
Dictionary Structure . . . 154
Memory Management . . . 154
Compile-time Features . . . 154
Index . . . 157

Chapter 1. Foreword

The idea of the Kingfisher Programming Language **(KPL)**, **(KF)** emerged from research into vintage Commodore computer systems, and the differences between them and contemporary systems.The result of this research is project Aves which considers the possibility of enhancing both hardware and software capabilities while adhering to the same architectural principles.

Project Aves includes new designs that incorporate the original video and sound devices with currently available devices, which offer lower power consumption, higher performance and additional capacity. To access these additional capabilities updated software is also required, and this is where Kingfisher fits in.

Commodore's original BASIC and Kernal had limited features to enable accessing disk drives, the custom video, and the sound chips. Kingfisher provides built in functions to achieve this, and a method to extend the language should the need arise.

The language itself draws inspiration from several sources: the stack-based simplicity of Forth, the readability of Python, the modularity of Go, and the type safety of Modula-2. These influences have been adapted to work within the limited resources of the target platforms while enhancing the programmer experience.

Kingfisher currently supports:

- Commodore 8-bit computers (PET, CBM series, VIC-20, C64, C16, Plus/4)
- Aves retro systems based on:
- Rockwell R65C02
    - WDC 65C02/65C81
    - NEC V25/35
    - Motorola 68HC000
    - Zilog Z16C01

Kingfisher's foundation is Forth, although very little of the standard Forth vocabulary is utilised. Forth's Dictionary, parameter stack, interpreter and compiler are retained, and the compiler produces threaded code. This architecture is well suited to vintage systems with limited resources, it is easy to implement, and its inherent simplicity allows it to be ported to different CPUs with relative ease. The inner interpreter (or virtual machine) can be built using Indirect, Direct, or Subroutine threading without requiring extensive code changes. The Text Interpreter and Compiler are also remarkably small and are factored in to small execution units called words which pass values via the parameter stack. All of these features are highly desirable and help to make Kingfisher compact, with enough flexibility to optimise the architecture to suit the underlying hardware.

In addition to these core Forth features Kingfisher introduces new features found in other languages such as strong typing, object orientation and additional data structures.

Kingfisher is a statically typed language which allows most of the checking to be done at compile time, and to simplify memory management collections and strings are passed by reference.

The object oriented programming features are based on Go's struct and linked method architecture, which are called Type and Method in Kingfisher. Data in the Type structure can only be accessed from a linked method which provides a mechanism for data hiding and encapsulation. Polymorphism is achieved by adopting Forth's stack effect comments as part of the method signature, and replacing Forth's comments with the more common // style. Types and methods are used to create abstract data types in Kingfisher to implement all of its variables and collections.

This reference document provides comprehensive documentation of Kingfisher's syntax, features, and implementation. It is intended that this document will be used as a blueprint for language implementation as part of a 'Working Backwards' philosophy, which starts with the final product and works backwards identifying the components required to deliver the final product or service.

Chapter 2. Abstract

Kingfisher is implemented as a stack-based architecture, where operations primarily manipulate a Last-In-First-Out (LIFO) stack. This architectural choice provides several advantages for resource-constrained systems, wwhich are a simplified instruction set, reduced memory overhead, and straightforward implementation of control structures. The stack serves as the primary means of passing parameters between operations, eliminating the need for complex parameter passing mechanisms and register allocation strategies. This approach, combined with strong type checking and an interactive development environment, makes Kingfisher particularly well-suited for vintage computing platforms and amateur built retro systems where memory and processing resources are limited.

A primary goal of Kingfisher design was to improve memory safety, and strong typing is the first element in achieving this goal, but memory allocation and bounds checking are also a high priority.

- **Strong Typing** Enables the compiler to check that the correct routine is being used for the data it is handling. During compilation all data being transferred via the stack is used to determine the correct code to compile.
- **Memory Allocation** The use of memory allocation is limited to types and modules, and there is no direct pointer manipulation. Template information representing variables and collections are added to the dictionary which grows upwards at compile-time. At runtime memory from the heap which grows down from the top is allocated to store the data. This approach allows code to be placed in ROM, and offers the possibility of dynamic allocation and release of memory when local variables go in and out of scope.
- **Bounds Checking** Both data and code objects are stored in the dictionary. All objects include code, data objects also include the data, pointers to data and meta data, which is used to check data boundaries.

Part I: Getting Started

Kingfisher requires minimal configuration and setup to begin operations. The system is available as a .D64 file for use with the VICE Emulator offering rapid deployment, as a ROM Cartridge, or as a download from the git repository.

This part of the manual presents the fundamentals of the Kingfisher programming language. The content covers system acquisition and loading procedures, operation of the interactive REPL (Read-Evaluate-Print Loop) environment, and the basic concepts that distinguish Kingfisher. The REPL environment facilitates immediate, interactive experimentation with language features, code testing, and practical understanding of the system's operation. The material is structured to provide essential foundations for both newcomers to programming and experienced developers, establishing the necessary groundwork for effective Kingfisher program development.

Chapter 3. The Kingfisher Language

Kingfisher is a stack-based programming language designed to bring contemporary programming concepts to classic computing platforms. While maintaining compatibility with Commodore systems through its .D64 disk image format, Kingfisher extends beyond traditional BASIC capabilities to offer a robust and flexible programming environment.

Design Philosophy

The language emphasises simplicity and power through its stack-based architecture, providing direct memory access while maintaining strong type safety. Built on fundamental computer science principles, Kingfisher combines the immediacy of interactive programming with the structure and capabilities expected in modern development environments.

System Architecture

The Kingfisher system architecture integrates a compact yet powerful kernel for managing system resources alongside an interactive development environment based on the REPL (Read-Evaluate-Print Loop). These core components work in conjunction with standard modules that provide essential programming tools and utilities for development.

During system initialisation, the kernel establishes the programming environment, presenting the user with an interactive prompt, as shown below.

```
>>> Kingfisher Programming Language V0.1
31744 Bytes Free
ok
```

The system adapts to available hardware resources, with memory allocation adjusted according to the installed capacity. A typical configuration with 32KB of RAM reserves 1024 bytes for system operations, leaving the remainder available for program storage and execution.

Chapter 4. Conventions and Standards

The Kingfisher programming environment employs consistent conventions for naming, documentation, and code formatting. These standards ensure code readability and maintainability while providing clear guidance for development practices.

Naming Standards

Kingfisher enforces strict naming conventions across different language elements to enhance code clarity and maintain consistency throughout programs. Each language element follows a specific case convention that clearly identifies its purpose and scope within the system.

Table 1. Naming Standards

Element	Convention	Example
Type	PascalCase	`Integer`, `StringBuffer`, `CustomType`
Method	PascalCase	`ReadLine`, `WriteBytes`, `ConvertToString`
Definition	PascalCase	`ProcessData`, `HandleInput`, `ValidateUser`
Variable	camelCase	`counter`, `firstName`, `bufferSize`
Constant	kebab-case	`max-buffer-size`, `default-timeout`, `screen-width`
Module	kebab-case	`max-buffer-size`, `default-timeout`, `screen-width`

Documentation Standards

Documentation in Kingfisher follows specific formatting conventions to ensure clarity and consistency across all reference materials. Code examples throughout the documentation demonstrate practical implementation of concepts and can be executed directly in the REPL environment.

Stack Position References

When working with stack operations, it's essential to have a clear way to reference different positions in the stack. Kingfisher uses a consistent labeling system throughout its documentation, particularly in operation descriptions and stack effect annotations. The following table defines these standard position references.

Table 2. Stack Position References

Label	Position	Description
T	First	Top of stack
N	Second	Next on stack
B	Third	Bottom of stack

Code Examples

All code examples appear in dedicated code blocks, clearly separated from the surrounding text. Examples include contextual comments that provide additional information without affecting code execution.

Code Block Example

```
// This is a comment in a code block
```

Technical References

The documentation uses several standardised reference formats. Stack Signatures provide clear display of parameter requirements and expected results. Constructor and Method tables present formal definitions of components. Type annotations within examples indicate specific data types. Cross-references throughout the documentation connect related concepts and provide detailed explanations where needed.

Chapter 5. The Interactive Development Environment

The Kingfisher REPL provides an interactive programming environment through its command-line interface. The system accepts input one line at a time, with immediate feedback after each command execution. This interactive environment facilitates rapid development, testing, and learning through direct experimentation with language features. The environment combines efficient command entry with comprehensive debugging capabilities, making it suitable for both development and educational purposes.

Chapter 6. Using the REPL

Once the `ok` prompt is displayed the REPL is ready to accept commands. After entering values and pressing ENTER a status message of `ok` or `ERROR` followed by a debug message is displayed. If the operation results in an error a message, is printed on the following line. Error messages can be found in this sectionc [Compilation and Runtime Errors]. The Debug message consists of the top three stack values with a [Type Suffix] and the current position of the stack pointer. A more detailed analysis of the stack can be found with `StackInfo` (see [Debugging in the REPL]).

Example REPL Session

```
1B 2 3S ok - 1B 2V 3S, SP=3
1 2 3B 4S ok - 2V 3B 4S, SP=4
7 6 * ok - 42V, SP=1
6 7 swap / Error - 6V 7V, SP=2
'swap' Not Found

+ Error, SP=0
Stack Underflow

7 6 * Error - 0B 7W 6W, SP=192
Stack Overflow

10 0 / Error - 10W 0W, SP=2
Divide by Zero
```

NOTE	Exact error messages may vary from these examples, consult Error Message Reference in part VI for further details.

The Command Line Interface (CLI)

The REPL is accessed via a CLI where commands are entered one line at a time, each line can be a maximum of 80 characters. The CLI maintains a 256 byte history buffer, which provides up and down arrows for scrolling through the buffer. For more details see the Perch CLI Reference in Part VI.

Comments in Kingfisher

Comments in Kingfisher are a word called // which must be space delimited like any other word. // can be used in the compiler and interpreter environments and ignores all content after the trailing space up to the end of the current line.

Chapter 7. Stack-Based Programming Fundamentals

Before diving into Kingfisher programming, it's essential to understand how the language handles calculations and data manipulation through its stack-based architecture. This section introduces the fundamental concepts of postfix notation and stack operations that form the foundation of Kingfisher's programming model.

Understanding Postfix Notation

Before we begin its important to note that Kingfisher uses Postfix (also known as reverse Polish notation) rather than the usual Infix notation. With postfix notation the operator follows both operands rather than in between them.

For example the infix calculation `3 * 3 - 3 / 3 + 3` evaluates to 5 if the calculation is performed by evaluating from left to right, and 11 if operator precedence rules are applied.

To standardise things a rule defining the precedence of operators was created by mathematicians in the 1600s. Since then it has been adopted for use in programming languages and is now the accepted norm. The rule is expressed as Brackets/Parenthesis, Order/Indices/Exponent, Divide, Multiply, Add, Subtract, which is shortened to BODMAS or BIDMAS in the UK, and PEMDAS in the US and Australia. (**Note:** the reversal of Multiply and Divide, this is not significant because they have equal weight.)

There is no such ambiguity with postfix notation however. If the calculation is written `3 3 / 3 3 * 3 + Swap -`, then the result can only be 11.

The results from intermediate calculations need to be stored temporarily, which is achieved using a LIFO `Stack`. The Swap operator before the - swaps the top two values on the stack so that the subtract operation is performed correctly (Order is significant for both the Divide and Subtract operations).

Postfix can feel a bit counter intuitive to begin with but allows for complex operations without having to use lots of brackets, and helps present an extremely clean and simple syntax. A commented example with a stack debug output showing the top 3 values of the stack is shown below.

Infix Notation, Left to Right

```
3 * 3 - 3 / 3 + 3

3 * 3 = 9
9 - 3 = 6
6 / 3 = 2
2 + 3 = 5
```

Infix Notation BODMAS

```
3 * 3 - 3 / 3 + 3

3 / 3 = 1
3 * 3 = 9
9 + 3 = 12
12 - 1 = 11
```

Postfix Notation

```
3 3 *   // 3 * 3 = 9
3 3 /   // 3 / 3 = 1
3 +     // 9 + 3 = 12
-       // 12 - 1 = 11
```

Chapter 8. Number System

Numbers in Kingfisher can be represented in three formats. Decimal numbers (base 10) are written directly without a prefix and can be positive or negative using the '-' prefix. Hexadecimal numbers (base 16) use the '0x' prefix and can contain digits 0-9 and A-F (case insensitive). Binary numbers (base 2) use the '0b' prefix and contain only 0 and 1 digits. Both hexadecimal and binary formats use two's complement for negative values.

Kingfisher is strongly typed, requiring type information for the compiler to check compatibility. When entering literals, a type suffix must be provided. Decimal numbers default to V type if no suffix is specified, while hexadecimal and binary numbers default to B type. For hexadecimal values, a separator is required before the type suffix to prevent ambiguity. Any operation that would result in a value exceeding the capacity of its declared type will raise an error. Variables must be constructed as a specific type before use and cannot change type once created. For a complete reference of supported data types and their ranges, see [Data Types and CPU Support] in Part VI.

Table 3. Number Suffixes

Suffix	Type	Description
V	Var	Value with same size as address bus
B	Byte	8 bit unsigned value
W	Word	16 bit signed value
S	Sword	24 bit signed value
L	Long	32 bit signed value
F	Flag	8 bit, 0=false, 1=true

Type Suffix Examples

```
1W 2W +         // Stack: 3W    (Word + Word -> Word)
1B 2B +         // Stack: 3B    (Byte + Byte -> Byte)
1W Dup          // Stack: 1W 1W (Scalar duplication)
3B 4B Compare   // Stack: 1F    (Scalar comparison)
```

Chapter 9. Error Management

Kingfisher implements a modern exception-based error handling system. When an error occurs, an exception is raised rather than returning error flags or codes. This approach ensures that errors cannot be accidentally ignored and maintains a clean separation between normal program flow and error handling.

Error Types and Handling

Errors in Kingfisher fall into two categories. The first category consists of catchable exceptions that can be handled by user code. These exceptions allow for graceful recovery and cleanup when errors occur during normal program operation. The second category comprises fatal errors that terminate program execution. These fatal errors typically indicate serious system-level problems and cannot be caught - they will always return control to the operating system.

For a complete reference of all error conditions and their handling, see Error Message Reference in Part 6.

Chapter 10. Stack Signatures

Kingfisher has taken Forth's stack notation and extended it to act as a type signature. This section describes the signature system and its components.

Type Categories

Stack signatures use several categories to indicate type requirements. These categories are for documentation purposes only and cannot be used to define a type, they are a used to represent words that have definitions with multiple signatures.

Category	Description
Pointer	Represents any pointer types such as Strings, Arrays, Slices etc.
Base	Base types are include all scalar types and String (Var, Byte, Word, Sword, Long, Flag, String)
g1, g2, etc.	Represent generic values that can be any type but provide positional information, useful when describing stack operations
Numeric	Represents any numeric type (Var, Byte, Word, Sword, Long)
Scalar	Represents any value of the built in scalar types (Var, Byte, Word, Sword, Long, Flag)

Chapter 11. The Parameter Stack

The parameter stack serves as the foundation for all computations in Kingfisher. It's important to note that Kingfisher uses a single 'Parameter Stack' for all operations. Kingfisher does employ separate stacks for subroutine calls and type checking, but these are not user-addressable and do not have operators.

Stack Signatures

Values passed on the stack are managed using stack signatures, which are used to identify parameter types, providing a consistent notation that clearly shows both inputs and outputs of an operation.

The signature is divided by a colon (:), where values before the colon represent inputs, and values after the colon represent outputs. The order of values in the signature corresponds directly to their position on the stack, with the rightmost value being the topmost stack item.

For example, in the signature (g1 g2 : g2 g1), g1 and g2 are inputs where g2 is on top of the stack, and after the operation, their positions are swapped with g1 now on top.

Example Stack Signatures

```
// Exchanges top two stack items
( g1 g2 : g2 g1 ) Swap

// Duplicates any scalar value
( g1 : g1 g1 ) Dup

// Removes any scalar value
( g1 : ) Drop

// Copies second item to top
( g1 g2 : g1 ) Over
```

Type Specific Signatures

```
// Calculate the bitwise And of two bytes
( Byte Byte : Byte ) And

//
( Var Var : Var ) Add
```

Stack Operators

Stack operators are used to reorder, duplicate, and remove values on the stack. All operations are fully polymorphic, and allow any mix of types.

Table 4. Stack Words

Operation	Description
(g1 :) Drop	Removes the top stack value
(g1 : g1 g2) Dup	Creates a duplicate of the top stack value
(g1 g2 : g2 g1) Swap	Exchanges the top two stack values
(g1 g2 : g1 g2 g1) Over	Copies the second stack value to the top
(g1 g2 g3 : g3 g1 g2) ror	Rotates the top three stack values to the right
(g1 g2 g3 : g2 g3 g1) rol	Rotates the top three stack values to the left

Stack Manipulation Examples

```
1 Dup ok 1 1 SP=2           // Duplicate top value
Drop ok 1 SP=1              // Drop the top value
1 2 Swap ok 2 1 SP=2        // Swap the top two values
1 2 Over ok 1 2 1 SP=3      // Move N to the top of the stack

// Rotate the top three values 1 position to the right
1 2 3 ror ok 3 1 2 SP=3

// Rotate the top three values 1 position to the left
1 2 3 ror ok 2 3 1 SP=3
```

Chapter 12. Arithmetic Operators

Arithmetic operators perform mathematical calculations on numeric values. These operations follow standard mathematical rules while enforcing type safety. All arithmetic operations require numeric types and will raise appropriate exceptions for type mismatches or mathematical errors such as division by zero. Results maintain the type of their operands, with overflow checking performed automatically.

Table 5. Arithmetic Words

Operation	Description
(Number Number : Number) *	Multiply N by T and leave result on the stack
(Number Number : Number) /	Divide N by T and leave the quotient on the stack
(Number Number : Number) %	Divide N by T and leave the remainder on the stack
(Number Number : Number Number) /%	Divide N by T and leave the quotient and remainder on the stack
(Number Number : Number) +	Add N to T and leave the sum on the stack
(Number Number : Number) -	Subtract T from N and leave the difference on the stack

Stack Manipulation Examples

```
2 3 * ok 6 SP=1     // Multiply 3 by 2
6 2 / ok 3 SP=1     // Divide 6 by 2
3 7 + ok 10 SP=1    // Add 3 to 7
10 8 - ok 2 SP=1    // Subtract 8 from 10

10 9 % - ok 1 SP=1  // Remainder (modulus) of 10 / 9

// Divide 10 by 9 return quotient and remainder
10 9 /% - ok 1 1 SP=2
```

Bitwise Operators

Bitwise operators provide low-level manipulation of individual bits within values. These operations are essential for systems programming, and hardware interfacing. All bitwise operations work on Byte values (0-255) and manipulate their binary representation directly. For shift operations, bits moved beyond the byte boundary (8 bits) are discarded, maintaining the byte-sized result.

Table 6. Bitwise Words

Operation	Description
(Byte Byte : Byte) And	Bitwise And of N with T
(Byte Byte : Byte) Or	Bitwise Or of N with T
(Byte Byte : Byte) Xor	Bitwise Xor of N with T

Operation	Description
(Byte Byte : Byte) <<	Shift N left by T bits
(Byte Byte : Byte) >>	Shift N right by T bits
(Byte : Byte) Not	Bitwise Not. Invert all bits

Stack Manipulation Examples

```
// And 170 with 240
0b10101010 0b11110000 And ok 160 SP=1

// Or 170 with 240
0b10101010 0b11110000 Or ok 160 SP=1

// Xor 170 with 240
0b10101010 0b11110000 Xor ok 80 SP=1

// Invert 170
0b10101010 Not ok 85 SP=1

// Shift left
0b10101010 << ok 84 SP=1

// Shift right
0b10101010 << ok 85 SP=1
```

General Operators

General operators provide essential functionality that complements the other operator categories. These operations include type conversion, absolute value calculation, and character handling. They maintain type safety while offering flexibility for common programming tasks. String conversion operators ensure safe transformation between string representations and numeric types, with appropriate error handling for invalid conversions.

Table 7. General Words

Operation	Description
(Number : Number) Abs	Converts T to a positive number
(Number Number : Number) Min	Returns the smaller of two numbers
(Number Number : Number) Max	Returns the larger of two numbers
(String : Number) ToNumber	Converts the string to a Number type
(Number : String) ToString	Converts the number to a String type
(Scalar : Byte) SizeOf	Returns type size in bytes

Stack Manipulation Examples

```
"Hello" Print     // Prints "Hello"
-1 Abs ok 1 SP=1    // return the unsigned value T
1 2 Min ok 1 SP=1   // return the lowest value
3 4 Max ok 4 SP=1   // return the highest value

// Convert value in string to number
"12" ToNumber ok 12 SP=1
"123B" ToNumber ok 123B SP=1

// convert number to string
123 ToString ok <String> SP=1
123W SizeOf ok 2 SP=1
```

Chapter 13. Built-in Types

Kingfisher has Scalar, String and Array types built in, which enables the construction of variables. Data held within a typed variable can only be accessed using the linked methods. These linked methods are customised to operate on the variable contents only, which ensures type compatibility

Scalar Types

The supported Scalar types are listed below in the [Supported Scalar Types]. A new variable is constructed using the type name followed by a unique name, as described in the Scalar variables examples.

Table 8. Scalar Constructors

Constructor	Description
Var	A value that is the same size as the address bus. See [Supported Architectures]. It can be Word, Sword, or Long. Used for index operations for arrays and buffers. Size matches CPU architecture
Byte	An 8 bit unsigned number, with a range of: 0-255. Used for character data, strings, text manipulation, IO operations
Word	A 16 bit signed number, with a range of: -32768 - 32767 Default choice for integer maths and real-time applications
Sword	A 24 bit signed number, with a range of: -8388608 - 8388607 Used for 20-24 bit addressing, efficient integer maths with better resolution
Long	A 32 bit signed number, with a range of: -2,147,483,648 - 2,147,483,647 Used for 32-bit addressing, high-resolution integer and fixed-point maths
Flag	An 8 bit wide type containing values of 0=False, and 1=True Used in all boolean and operations

Chapter 14. Constants

Constants provide a way to associate meaningful labels with values, replacing literal values in code. Constants are immutable once defined and can be used with any supported data type. The compiler optimises constant usage, resulting in smaller code footprint and improved performance.

Declaring Constants

Constants are declared using the 'Constant' keyword following the value and type (if specified). By convention, constant names use uppercase letters or kebab-case.

Constants are commonly used for character codes and control characters, status flags and boolean values, mathematical constants, configuration values, and state definitions. Using constants instead of literal values improves code readability by providing meaningful names and ensures a single point of maintenance for frequently used values. This approach reduces typing errors, enables better code navigation and refactoring support, and allows the compiler to perform type checking at compile time while optimising for a smaller code footprint.

Examples

```
32B Constant spc
8B  Constant tab
10B Constant nl
13B Constant cr

0  Constant zero
1  Constant one
-1 Constant neg-one

0F Constant true
1F Constant false

true Print      // Prints 1
zero Print      // Prints 0
```

Chapter 15. Variables

At the heart of Kingfisher's type system lies its approach to variable management. While the stack provides temporary storage for passing values between functions, variables offer persistent state storage that can be accessed at any time within their active scope. Unlike many traditional programming languages that permit dynamic typing, Kingfisher implements a rigorous type-safe environment where variables must be explicitly declared with their intended type before use.

Variable Construction

Variables are constructed and instantiated using specific type constructors. Each variable is initialised to 0 by default, ensuring a known starting state. This type-safe approach ensures that variables maintain their designated type throughout their lifetime, preventing type-related errors during program execution.

In addition to the standard constructor there is a literal constructor which is used to set the initial value of the variable.

Table 9. Variable Constructors

Constructor	Description
(:) Var name	Constructs a variable called name, sized to match the architecture's address bus width
(:) Var(name number)	Constructs a variable called name sized to match the architecture's address bus width, and initialises it with number
(:) Byte name	Constructs an 8-bit unsigned variable called name
(:) Byte(name number)	Constructs an 8-bit unsigned variable called name, and initialises it
(:) Word name	Constructs a 16-bit signed variable
(:) Word(name number)	Constructs a 16-bit signed variable called name, and initialises it with number
(:) Sword name	Constructs a 24-bit signed variable
(:) Sword(name number)	Constructs a 24-bit signed variable called name, and initialises it with number
(:) Long name	Constructs a 32-bit signed variable
(:) Long name(name number)	Constructs a 32-bit signed variable called name, and initialises it with number
(:) Flag name	constructs an 8 bit unsigned boolean variable called name. flag can only contain 0-false, 1-true
(:) Flag(name 0	1)

Variable Constructor Example

```
Byte shortCard      // Create a byte value set to 0
Byte( test 100 )    // Create a byte value set to 100
```

Variable Operations

Once constructed, variables support a consistent set of operations through method calls. These operations provide a clean interface for reading, modifying, and managing variable state. All operations maintain type safety and perform bounds checking to prevent invalid states.

Table 10. Variable Methods

Operation	**Description**
(Scalar :) Add	Performs bounds-checked addition of the given value to the variable's content. Not available with the Flag type
(: Scalar) Get	Retrieves the current value while maintaining type safety
(:) Reset	Reinitialises the variable to its default state of 0
(Scalar :) Set	Assigns a new value after performing type compatibility verification
(Scalar :) Sub	Executes bounds-checked subtraction from the variable's current value. Not available With the Flag type
(:) Toggle	Toggle the value of the flag. Flag type only

Scalar Variable Operations

```
// Variable construction and initialisation
// Creates a 16-bit signed counter
Word Variable counter

// Creates an 8-bit unsigned status indicator
Byte Variable status

// Creates a boolean state flag
Flag Variable ready

// Value manipulation
// Assigns 42 to counter
42 counter.Set

// Displays current counter value
counter.Get Print

// Increases counter by 5
5 counter.Add

// Returns counter to 0
counter.Reset
```

Variable Interactions

```
// Demonstrating type safety and bounds checking
0xFF00 Constant MAX-VALUE
297S Constant OFFSET
0xAAB Constant MASK

Var total
Byte value

// Sets total to maximum value
MAX-VALUE total.Set

// Reduces total by offset value
OFFSET total.Sub

// Assigns mask to value
MASK value.Set

// ERROR: Type mismatch
value.Get total.Add

// Adds value to total with bounds checking
value.Get toVar total.Add
```

Chapter 16. Collections

In Kingfisher's memory model, collections provide structured storage for multiple values, extending beyond the single-value limitations of variables. The language implements collection types to address different data storage needs: Arrays for sequential numeric values and Strings for text manipulation. Arrays are available for each Scalar Type, providing type-safe storage and operations.

Chapter 17. Arrays

Arrays provide fixed-size sequential storage for values of uniform type. Unlike variables which hold single values, arrays en able developers to manage related data as a cohesive unit, supporting both direct element access and slice operations for working with subsequences. Arrays are available for each Scalar Type, ensuring type safety across all operations.

Table 11. Array Constructors

Constructor	Description
(Var :) Array name	Creates a fixed-size array called name of type Var with specified length
(:) Array[name number1 number2 … numberN]	Creates a generic array called name from the succeeding literals
(Var :) <T>Array name	Creates a fixed-size typed array called name with specified length
(:) <T>Array[name number1 number2 … numberN]	Creates a typed array from a literal sequence of the same type

NOTE <T> is one of Var, Byte, Word, Sword, Long and Flag

Array Slices

Array slices provide a lightweight mechanism for working with subsequences of arrays without copying data. They maintain a reference to the underlying array along with bounds information, enabling efficient operations on partial sequences. Like their parent arrays, slices are available for both generic and typed arrays.

Table 12. Array Methods

Operation	Description
(Var Var :) Add	Add value of N to value element T.
(ArraySlice Var :) Copy	Copies an Slice from pointer in N, size T into this array at position 0
(ArraySlice Var Var :) Copy	Copies an Slice from pointer in B, size N into this array at position T
(ArraySlice Var Var :) Fill	Copies Slice from base pointer B, length N elements, with value T
(: Var) Length	Get Length of Array in T
(Var :) Get	Get value of element T
(Var :) Reset	Reset element N to zero
(Var Var : ArraySlice Var) Slice	Creates a slice view of array elements between start N, and end T indices. Returns pointer to slice N and length T

Operation	Description
(Var Var :) Set	Set the value in element T to the value in N
(Var Var :) Sub	Subtract N from the value in element T

Table 13. Typed Array Methods

Operation	Description
(<T>Slice Var :) Copy	Copies a Slice from pointer in N, size T into this array at position 0
(<T>Slice Var Var :) Copy	Copies a Slice from pointer in B, size N into this array at position T
(<T>Slice Var Var :) Fill	Copies Slice from base pointer B, length N elements, with value T
(Var :) Get	Get value of element T
(Var :) Reset	Reset element T to zero
(Var Var : <T>Slice Var) Slice	Creates a slice view of array elements between start N (inclusive), and end T (exclusive) indices. Returns pointer to slice N and length T
(Scalar Var :) Set	Set the value of element T to the value in N
(Scalar Var :) Sub	Subtract the value of element N to the value in N. Not FlagArray
(Var :) Toggle	Toggle the value of element T. FlagArray only

Array Examples

```
100 Array test
20 FlagArray flags

Array[ primes 2 3 5 7 11 13 17 19 ]

// Sets the value of element 50 to 1
1 50 test.Set

// Gets the value from element 50 (=1)
50 test.Get

// Adds 1 to the value in element 50 (=2)
1 50 test.Add

// reset element 50 to zero
50 test.Reset

// Fill array from element 10 with 10 copies of 5555 hex
10 10 test.Slice 0x5555/V test.Fill

// Copy primes to beginning of array
0 8 primes.Slice test.Copy

// Copy primes to element 10 of array
0 8 primes.Slice 10 test.Copy
```

Chapter 18. Strings

Strings serve as fixed-length buffers optimised for text storage and manipulation. A single byte length is used to constrain the length of the string to between 0 and 255. The implementation ensures bounds checking on all string operations.

Table 14. String Constructors

Constructor	Description
(Byte :) String name	Constructs a fixed-length string buffer called name with specified length in T
(:) String(name "string")	Constructs an immutable string called name containing the literal string

String Constructor Example

```
// Create an immutable string
String( hello "Hello, World" )

25 String name      // Create a string buffer to hold a name
```

String Slices

String slices provide a lightweight mechanism for working with subsequences of strings without copying data. They maintain a reference to the underlying string buffer along with bounds information, enabling efficient text operations on partial sequences.

Table 15. String Methods

Operation	Description
(String :) Append	Appends String T to end of current String truncating at length of buffer
(StrSlice Byte:) Append	Appends StrSlice N, of length T to end of current String truncating at length of buffer
(StrSlice Byte :) Copy	Copies slice contents to string buffer position 0
(StrSlice Byte Byte :) Copy	Copies slice contents to string buffer at given position
(String : StrSlice Byte) Find	Find Substring T in this string and return StrSlice N length T
(: String) Get	Returns a String reference
(Byte : Byte) Get	Returns the character at element T
(: Byte) Length	Returns the length of the string
(: Byte) Size	Returns the size of the string buffer
(String :) Set	Sets the string Reference to point to string T

Operation	Description
(Byte Byte :) Set	Sets the the value of element T to character N
(Byte Byte : StrSlice Byte) Slice	Creates a string slice view between start N (inclusive) end T (exclusive)
(StrSlice Byte :) Lower	Converts all characters in the slice to lowercase
(StrSlice Byte :) Upper	Converts all characters in the slice to uppercase
(StrSlice Byte :) Title	Converts all character in the slice to title case
(String :) Trim	Removes leading and trailing whitespace from Trim

Collection Operations Example

```
// Demonstrating collection operations
20 String message

// Store greeting in string buffer
String( hello "Hello, World" )

0 hello.Length hello.Slice message.Copy

// Convert to uppercase
message.Upper

50 Array numbers

// Initialise an array with a sequence
String[ num 1 2 3 4 5 ]
0 Array.Length num.Slice numbers.Copy

// Display subset of array elements
2 4 numbers.Slice Print
```

Chapter 19. StrArrays

StrArrays provide fixed-size sequential storage for multiple strings, enabling efficient management of text collections. Each element in a StrArray can store a string of up to 255 bytes, following the same capacity rules as individual String types (see Strings). StrArrays support both direct element access and slice operations for working with subsequences.

Table 16. StrArray Constructors

Constructor	Description
(Var Var :) StrArray name	Constructs a fixed-size array, size N, of strings with specified length T
(:) StrArray[name "string1" "string2" … "stringN"]	Constructs an immutable array of strings, from specified list of literal strings

StrArrSlices

StrArrSlices provide a lightweight mechanism for working with subsequences of string arrays without copying data. They maintain a reference to the underlying string array along with bounds information, enabling efficient operations on partial sequences.

When copying between slices and arrays, the `Copy` operation performs a deep copy, meaning it duplicates both the array structure and the contents of each string. This ensures that modifications to strings in the source array won't affect the destination array.

Table 17. StrArray Methods

Operation	Description
(StrArrSlice Var :) Copy	Copies T elements from StrArrSlice N to StrArray starting at position 0. Performs deep copy of String contents.
(StrArrSlice Var Var :) Copy	Copies N elements from StrArrSlice B to element starting at T, bounded by the array length
(Var : String) Get	Returns a reference to the String at element T
(String Var :) Set	Sets element at T to the String reference N
(Var Var : StrArrSlice Var) Slice	Creates a slice view from index N (inclusive) to index T (exclusive). Returns StrArrSlice with base pointer N and length T

StrArray Operations Example

```
// Create array of 5 strings length 15
15 5 StrArray buffer

// Create array of strings length 15 from literal StrArray
StrArray[ names "Alice" "Bob" "Carol" ]

// Slice whole Array and copy to Buffer starting at element 2
0 3 names.Slice 2 buffer.Copy

// Create another StrArray for John
StrArray[ john "John" ]

// Copy to element 0 in buffer
0 1 john.Slice buffer.Copy

// Set position 1 to point to String literals
"Arnold" 1 buffer.Set

// Get Reference from position 1 and print Arnold
1 buffer.Get Print
```

Chapter 20. Definitions

Programming languages provide various ways to structure and organise code into reusable units. In traditional languages, these might be functions, procedures, or methods. Kingfisher approaches this through the concept of word definitions, drawing inspiration from stack-based languages while providing modern programming constructs.

A word in Kingfisher represents a named sequence of operations that can be executed as a single unit. Each word encapsulates specific functionality, making programs more modular and easier to understand. When defining a word, developers specify both its interface through a stack signature and its behaviour through a sequence of operations.

The system processes words in two distinct phases. During the definition phase, it creates a dictionary entry and validates the word's composition. When the word is later invoked, the system opens a local scope, executes the compiled operations, and then closes the scope upon completion.

Each definition should maintain a clear purpose and predictable stack behaviour, as can be seen from all of the examples shown below all variables are declared in the global scope. For information about working with local scope, see the [Scope] section.

Table 18. Definitions

Operation	Description
(Signature :) Def name	Start of a definition that creates a new word and opens a new scope. The signature is created and managed by the Signature construct
End	Required keyword that concludes the word definition

Definition Example

```
( Byte Byte : Byte ) Def Add
    + Print
End

5 2 Add      // Displays: 7
```

Variable Management Example

```
// Create byte variable
Byte Variable count

( Byte : ) Def Increment
    count.Get +     // Add input to current value
    count.Set       // Store result
End

5 count.Set         // Initialize to 5
2 Increment         // Add 2 to count
count.Get Print     // Displays: 7
```

Working with Strings

String manipulation can be encapsulated within word definitions. The following example shows how to work with string buffers.

String Operation Example

```
20 String message    // Create string buffer

( : ) Def ShowMessage
    String( hello "Hello" )

    hello.Get Print         // Print the literal directly

    // Copy "Hello" into buffer
    0 hello.Length hello.Slice message.Copy

    message.Get Print       // Display result
End

ShowMessage                 // Displays: HelloHello
```

Array Operation Example

```
// Create array of size 5
5 Array numbers

( Byte : ) Def StoreAndShow
    // Store value at position 0
    0 numbers.Set

    // Display first element
    0 numbers.Get Print
End

42 StoreAndShow    // Displays: 42
```

Complex Definition Example

```
Byte total
Byte count

( Byte : ) Def AddToTotal
    total.Add       // Add value to total
    1 count.Add     // Increment counter
End

total.Reset         // Clear total
count.Reset         // Clear counter

5 AddToTotal        // Add first value
3 AddToTotal        // Add second value

total.Get Print     // Display total: 8
count.Get Print     // Display count: 2
```

Part II: Language Fundamentals

Kingfisher provides robust mechanisms for organising code and managing program execution. These mechanisms include modules, includes, and comprehensive scope management that together create a flexible yet controlled environment for program development and execution.

Chapter 21. Program Organisation

The primary mechanism for organising programs are vocabularies. Vocabularies are the fundamental containers used to organise and manage words. These containers are stacked, with upper vocabularies taking precedence over those below. This stacking mechanism enables temporary shadowing of names while maintaining access to the broader program structure, supporting local definitions within methods, type-specific operations, and control structure scoping.

Bootstrap

Program initialisation forms the foundation of any programming language implementation. Kingfisher's bootstrap mechanism provides a clear and consistent way to start program execution, ensuring proper system initialisation and module loading while maintaining strong typing and scope management.

Bootstrap Operations

The bootstrap operations provide mechanisms to initialise programs in Kingfisher. These operations handle program loading, execution, and automatic startup configuration. Programs must begin with a program declaration, which is a header containing program entry points and address location information.

The load process uses the information in the header to update system variables before execution begins.

Table 19. Bootstrap Words

Operation	Description
(:) Auto name	Configures the existing definition, definition-name to run automatically at system startup. definition-name can be a loader, runner or definition in a ROM
(Var :) Deregister name	Removes the program header at the specified address from the current vocabulary.
(Var Var :) Header name	Creates a relocation header for a program image to be loaded at address N, and with an execution address of T
(:) Load("filename" [name])	Clears the current program and loads a new program from the specified file into memory. An optional header can be specified in name, to load the program to a specific location
(:) New	The vocabulary for the current programs and all of its contents are deleted. Any vocabularies created since the program was loaded will also be deleted
(:) Program(program-name entry-point)	Program must be used at the top of a program file to create a new vocabulary for the program and set the entry point

Operation	Description
(:) Run	Executes the currently registered program
(:) Save("filename" [header])	Saves the binary representation of the program to the specified file preceded by the specified header (optional). If the header is ommited the program is saved at the default memory location

Bootstrap Examples

```
// Set program name at top of file
"Calculate" Program< Calculator

// program code
// ---------------------------------

// Clear the current program (Optional)
New

// Load the calculator program
Load( "calculator.kf" )

// Execute the current program
Run

( : ) Def LoadRun
    // Load the calculator program
    Load( "calculator.kf" )

    // Execute the current program
    Run

// Set LoadRun to execute at boot
Auto LoadRun
```

Vocabularies and Chains

Vocabularies are collections of related word definitions that can be managed as a unit. They provide a way to organise and control access to definitions, similarly to modules but with more dynamic control. Chains are ordered lists of vocabularies that determine how words are found during program execution.

Chains

The chain system manages word lookup and visibility through four distinct chains, as shown in the [Vocabulary Chains] table.

The chain system uses a singleton object to manage an ordered collection of vocabularies. The object's linked methods are shown in the [Chain Operations] table.

Table 20. Vocabulary Chains

Chain	Description
Active	The main scope for active vocabularies. Used for both searching existing words and adding new word definitions.
Extended	Searched before the Active scope. Used for word lookups only; new definitions are always added to the Active scope.
Restricted	Used for limiting word lookup to a specific vocabulary.
Excluded	Contains vocabularies that are no longer being used, and have been removed from other chains.

Table 21. Chain Methods

Operation	Description
(:) Activate name	Moves the vocabulary at the top of the specified chain to the top of the Activated Chain
(:) Extend name	Moves the vocabulary at the top of the specified chain to the top of the Extended chain
(:) Restrict name	Moves the vocabulary at the top of the specified chain to the top of the restricted chain
(:) Exclude name	Moves the vocabulary at the top of the specified chain to the top of the Excluded chain
(:) Delete name	Deletes the vocabulary from the top of the specified chain

Vocabularies

A vocabulary is a named collection of word definitions. Vocabularies are used with chains to control word visibility.

Table 22. Vocabulary Constructor

Operation	Description
(:) Vocabulary name	Creates a new vocabulary with the given name

Vocabulary Example

```
// Create new vocabulary at the top of the active chain
Vocabulary MyWords

// Move vocabulary between chains
chain.Exclude Active        // Excludes MyWords
chain.Activate Excluded     // Activates MyWords
chain.Extend Active         // Extends MyWords
chain.Restrict Extended     // Restricts MyWords
chain.Delete Restricted     // Delete MyWords
```

Modules

Modules provide the primary organisational structure in Kingfisher, allowing code to be organised into reusable, self-contained units. Each module creates its own scope, preventing naming conflicts between different parts of a program.

Table 23. Module Words

Operation	Description
(:) End	Marks the end of the vocabulary and excludes it
(String :) Include	Compiles contents from the specified file into the current module, which can be overridden by a module statement in the included file
(:) Module name	Creates a new module with the given name
(:) Use name	Adds the named vocabulary to the extended list. The named vocabulary must currently be in the excluded list

Module Example

```
// Define new module
Module calculator

    // Module content
    ( Num Num : Num ) Def Add
        +
    End

    ( Num Num : Num ) Def Subtract
        -
    End
End

"calculator.kf" Include     // Includes the contents

// Accessing words defined in modules
calculator/Add

// Add module to context temporarily
Use calculator              // Add calculator to the extended chain
    Add
End                         // moves calculator back to the extended chain
```

Aliases

Aliases provide alternative names for existing words. They create a new reference to an existing definition within the specified scope. When creating an alias, the source name must already exist, and the new alias name must be unique within the specified scope. Aliases preserve all properties

of their source and can be used anywhere the source can be used. It is not possible to create aliases of aliases, as this would create indirect references that could become invalid.

Table 24. Alias Word

Operation	Description
(String :) Alias name	Creates a new reference named 'name' pointing to the word on the stack

Alias Example

```
+ Alias Add     // Plus now refers to Add
3 4 Add         // Same as 3 4 +

// Create type alias
Number Alias Integer            // Integer now refers to Number type

Module test
    Var testVar
End

"test/testVar" Alias testVar   // Module Alias
```

Scope and Lifetime Rules

The scope and lifetime of declarations in Kingfisher are determined by where they are defined in the program. There are three main scoping levels to consider: global scope for system-wide declarations, module-level scope for module-wide declarations, and definition-level scope for local variables and temporary storage. This scoping system provides a clean and predictable environment for managing names and storage while maintaining flexibility across global, module-wide and local definitions, ensuring clear visibility rules and predictable memory management.

Global Scope

Declarations in global scope are visible throughout the entire program, regardless of module boundaries. The global scope contains system vocabularies that provide core functionality, global constants that can be accessed from any module, and all definitions imported from modules.

Module Level Scope

Module level scope contains declarations that are visible throughout a single module. this includes word definitions, type definitions, constants, and aliases defined within the module. These declarations are accessible to all words within the module, and are available to other modules using the name/word-name fully qualified name, or imported via the Use word. This mechanism provides a clean namespace for module-specific functionality while preventing unintended name conflicts between modules.

Definition-Level Scope

Local variables and temporary storage exist only within their defining word. this localised scope ensures clean variable management by automatically cleaning up storage when a word completes, provides predictable memory usage through clear allocation and deallocation points, and protects against naming conflicts by keeping variables isolated within their defining word.

```
10W Constant value      // Global Value

Module example
    // Define a word to calculate the square of a number
    ( word : Word ) Square
        Word result      // Local variable

        Dup *           // Temporary stack values
        result.Set      // Store in local variable
        result.Get      // Retrieve local value
    End

    // Test the Square word
    ( : ) Test
        value Square    // Calculate the square of value
        result.Get      // Error: Unknown word
    End
End
```

Error Handling

Errors are generated by `err-code Raise`, which can be caught using a surrounding `OnError Catch` block or by referencing the `err` object.

Table 25. Error Handling Words

Operation	Description
(:) Catch	Marks the beginning of the error exception handler
(:) End	Resets the system error variable and exception handler, then resumes execution.
(:) OnError	Starts an error handling block, by setting the exception handler to point to the upcoming `Catch` word
(Byte :) Raise	Sets the system error variable to the value of T and raises an exception

NOTE Error handling can be found in later sections, and can only be used inside a defining word, see sections Definitions and Linked Methods for more information on defining words.

Chapter 22. Type Definitions and Linked Methods

In Kingfisher, user defined types and their linked methods work together to create objects that manage and protect data through a controlled interface. At the heart of each type is a data schema that defines the structure's fields, while linked methods provide the sole mechanism for accessing this data. this approach ensures data integrity by making all fields private, with access permitted only through the type's methods. These methods operate using a specialised vocabulary containing the primitives needed for field access. When declaring a type instance, the final size can be specified, providing flexibility in memory allocation.

Creating a type involves three distinct phases, each operating within its own scope and providing access to specific words needed for that particular stage. The process begins with defining the data schema, which establishes the structure and layout of the type's fields. Once the schema is complete, linked methods are created to provide controlled access to these fields, implementing the type's behaviour and ensuring data integrity. Finally, the type is instantiated, allocating memory and initialising the data structure for use.

Type Definitions

Types are defined in three categories: Scalar, ShortSeq, and Sequence. Each category has its own construction mechanism, and they all support linked methods, which are in a scope that is linked to the type entry.

Table 26. Type Definition Words

Operation	Description
(:) Scalar name	Creates a new Scalar type called name, and extends the scalar-type vocabulary
(Byte :) ShortSeq name	Creates a new ShortSeq type called name and extends the short-seq-type vocabulary
(Var :) Sequence name	Creates a new Sequence type called name and extends the seq-type vocabulary

Table 27. Scalar Vocabulary Words

Operation	Description
(Byte :) : name	Compiles the size T into the definition for the field name
(:) End	Completes the type definition and excludes the scalar-type vocabulary

Table 28. ShortSeq Vocabulary Words

Operation	Description
(Byte Byte :) : name	Compiles the size N and type T into the definition for the field name

Operation	Description
(: Byte) Byte	Returns the size of Byte (1)
(:) End	Completes the type definition and excludes the short-seq-type vocabulary
(: Byte) Long	Returns the size of Long (4)
(: Byte) Sword	Returns the size of Sword (3)
(: Byte) Var	Returns the size of Byte (2,3 or 4 dependent on CPU architecture)
(: Byte) Word	Returns the size of Byte (2)
(: Byte) Byte	Returns the size of Byte (1)

Table 29. Sequence Vocabulary Words

Operation	Description
(Var Byte :) : name	Compiles the size N and type T into the definition for the field name
(: Byte) Byte	Returns the size of Byte (1)
(:) End	Completes the type definition and excludes the sequence-type vocabulary
(: Byte) Long	Returns the size of Long (4)
(: Byte) Sword	Returns the size of Sword (3)
(: Byte) Var	Returns the size of Byte (2,3 or 4 dependent on CPU architecture)
(: Byte) Word	Returns the size of Byte (2)
(: Byte) Byte	Returns the size of Byte (1)

Type Definition Examples

```
// definition for Qword
( : ) Qword
    8 : value
End

( : ) Scalar
// Fixed size small FIFO
( : ) Scalar FiFo
    1 Byte : rxPtr      // FIFO receive pointer
    1 Byte : txPtr      // FIFO transmit pointer
    256 Byte : buffer   // Circular buffer
End

// Variable size small FIFO
( Byte : ) ShortSeq Fifo
    1 Byte : rxPtr      // FIFO receive pointer
    1 Byte : txPtr      // FIFO transmit pointer
      Byte : buffer     // Circular buffer
End
```

Linked Methods

Linked Methods provide a safe and uniform way to access the data fields. They are defined similarly to word definitions but are bound to a specific type. The syntax follows the pattern `stack-signature name Method method-name`, creating a new word that is linked to the specified type.

When creating a linked method, the system follows a precise sequence. First, it parses the stack signature and validates the type name. Then it creates a new dictionary entry and establishes the link between the method and its type. A new method scope is opened, allowing the compiler to process the word list. Finally, the scope is closed, completing the method definition.

Type Field Vocabulary

Kingfisher provides a minimal but powerful vocabulary for accessing and manipulating type fields within methods. this vocabulary leverages compile-time polymorphism to ensure both safety and efficiency. Field access is resolved at compile time, eliminating runtime type checking overhead while maintaining type safety. The vocabulary includes operations for type interpretation and bulk operations that make use of field size information available at compile time.

Table 30. Type Field Words

Operation	Description
(g1 : Scalar) As name	Changes type interpretation of value to named type
(Var : Byte) Field name	Transforms base address to address and length of named field

Operation	Description
(Var : Var Var) Field name	Transforms base address to address and length of named field
(Var : Byte) Size name	Returns the size of the names type in bytes
(: Var) this	Returns the base address of the current object in the heap

NOTE Additional memory access words are defined in System Primitives

Linked Methods Example

```
// Variable length buffer type
( Var Byte : ) Sequence Buffer
    // Current length
    1 Var   : length

    // Variable length data
    Byte    : data
End

// Get current length
( : Var ) Buffer Method Length
    // get the address of the field
    this Field length Drop
    // Get the length field using primitives
End

// Set new length
( Byte : ) Buffer Method SetLength
    // get the address of the field
    this Field length Drop
    // Set the buffer length field using primitives
End

// Get data as Array
( : ByteArray ) Buffer Method Data
    //
    this Field data Drop

    // Tell the compiler return type
    As ByteArray
End

// Store string in buffer
( String : ) Buffer Method SetString
    // Get buffer address
    this Field data
    // Copy string into buffer
End
```

Constructors and Destructors

Construction and Destruction is automatic. Types can define Initialise and Terminate methods that will be called after construction and before destruction respectively.

Initialisation and Termination Examples

```
( : ) TypeName Method Initialise
    // Called after construction
    // Set up initial state
End

( : ) TypeName Method Terminate
    // Called before destruction
    // Clean up resources
End
```

Method Execution

Methods are accessed using dot notation, where the instance name and method name are separated by a dot (instance-name.method-name). When executing a method, the interpreter first separates the instance name from the method name. It then locates the appropriate instance scope and finds the method within that scope. Finally, it executes the compiled code associated with the method.

Type Field Examples

```
( Var : ) Type Buffer
    1 Var : length  // Buffer length
    Byte  : data    // Variable length data
End

( StrSlice Byte : ) Buffer Method Copy
    // Get data field size and address
    this Field data

    // Clip the length to be copied
    Raw Min

    // Copy the data from slice  into field
End

// Constructor to fill buffer with zero
( : ) Buffer Method Create
    // Get data field size and address
    0x00 this Field data

    // Fill the field with zero
End
```

Datasets

A dataset is an immutable collection of different data types, for example a dataset can be used to represent reference data containing mixed text, numeric and boolean data. The dataset is made up

of variable length tuples and can read as an iteration, or via a numeric index.

Table 31. Dataset methods

Operation	Description
(:) Dataset name	The data set constructor defines a new data set with the specified name
(:) (	Starts the definition of a new record within the dataset
(:))	End the record definition
(:) End	Ends the dataset definition
(: Flag) IsLast	Returns true if the last value was returned from the end of the iteration
(Byte : any) GetItem	Returns the value of the item number specified in T
(: Scalar) Next	Returns the next record from the dataset
(:) First	Returns the first record from the dataset

Dataset example

```
Dataset NameAndAddress
    // Raffle price winners
    ( "John Smith" 101 "01234567890" )  // Name, ticket number and phone number
    ( "Jane Doe" 235 "09876543210" )
End
```

Chapter 23. Control Flow

Kingfisher program execution is controlled with Branching and Looping commands. Both branches and loops require the ability to test for the correct conditions before a decision on which instruction will be executed next using conditional operators as described in the following section.

Chapter 24. Boolean Operations

In Kingfisher, rather than having a large number of boolean operators as primitive operations, you can use `And`, `Or` and `Not` in combination with the language's basic comparison operators (like <, >, =) to produce Xor, Nand, Nor etc. These derived boolean operators allow you to perform logical operations in your programs.

Table 32. Boolean Words

Operation	Description
(Flag Flag : Flag) And	Returns the result of N And T False And False ⇒ False False And True ⇒ False True And False ⇒ False True And True ⇒ True
(Flag Flag : Flag) Or	Returns the result of N Or T False Or False ⇒ False False Or True ⇒ True True Or False ⇒ True True Or True ⇒ True
(Flag : Flag) Not	Returns Not T True ⇒ False False ⇒ True

Example

```
// Definition signature: takes 2 flags, returns 1 flag
( Flag Flag : Flag ) Def Xor
    // Duplicate top 2 values and AND them
    Over Over And

    // Duplicate top 2 values and OR them
    Over Over Or

    // AND the results and NOT the final result
    And Not
End

// True   (different values => true)
True False Xor

// True   (different values => true)
False True Xor

// False  (same values => false)
False False Xor

// False  (same values => false)
True True Xor
```

Boolean Combinations Examples 1

```
// Basic Combinations
( Flag Flag Flag : Flag ) Def AndWithNot
    // NOT first flag, AND with second flag
    Not Swap And
End

( Flag Flag : Flag ) Def OrWithNot
    // NOT both flags, then OR them
    Not Swap Not Or
End

( Flag Flag Flag : Flag ) Def MultiAnd
    // AND all three flags
    And And
End

( Flag Flag Flag : Flag ) Def MultiOr
    // OR all three flags
    Or Or
End
```

Boolean Combinations Examples 2

```
// Common Boolean Patterns
( Flag Flag : Flag ) Def Nand
    // AND the flags, then NOT the result
    And Not
End

( Flag Flag : Flag ) Def Nor
    // OR the flags, then NOT the result
    Or Not
End

( Flag Flag : Flag ) Def Implies
    // NOT first flag, OR with second flag
    Not Swap Or
End
```

Example Usage

```
// Example usage:
True False True AndWithNot  // False
True False OrWithNot        // True
True True True MultiAnd     // True
False True False MultiOr    // True

True True Nand              // False
True False Nand             // True
False True Nor              // False
True False Implies          // False
```

Conditional Operations

Kingfisher provides three fundamental conditional operators (<, >, =) that form the basis for all comparison operations. Each operator consumes two values from the stack and returns a flag. These basic conditional operators can be combined to create more complex comparisons. Common programming patterns like range checks, equality comparisons, and boundary tests can all be constructed from these fundamental operations.

Table 33. Conditional Words

Operation	Description
(Num Num : Flag) <	Result is true if N is less than T 3 < 5 ⇒ True 5 < 3 ⇒ False 3 < 3 ⇒ False

Operation	Description
(Num Num : Flag) >	Result is true if N is greater than T 3 > 5 ⇒ False 5 > 3 ⇒ True 3 > 3 ⇒ False
(Num Num : Flag) =	Result is true if N is equal to T 3 = 5 ⇒ False 5 = 3 ⇒ False 3 = 3 ⇒ True

Range Checks

```
// Definition to check if value is between bounds (exclusive)
( Var Var Var : Flag ) Def Between
    // Stack: (low x high -- flag)
    Swap Over >     // low > x
    Rcw Rcw <       // x < high
    And             // Combine conditions
End

// Definition to check if value is between bounds (inclusive)
( Var Var Var : Flag ) Def BetweenInc
    // Stack: (low x high -- flag)
    Swap Over 1 - > // (low-1) > x
    Rcw Rcw 1 + <   // x < (high+1)
    And             // Combine conditions
End

// Definition to check if value is outside range
( Var Var Var : Flag ) Def Outside
    // Stack: (low x high -- flag)
    Swap Over <     // low < x
    Rcw Rcw >       // x > high
    Or              // Combine conditions
End
```

Example Usage

```
1 5 10 Between      // True  (1 < 5 < 10)
1 1 10 Between      // False (1 not < 1)
0 5 100 BetweenInc  // True  (0 <= 5 <= 100)
0 0 100 BetweenInc  // True  (0 <= 0 <= 100)
1 0 10 Outside      // True  (0 < 1 OR 0 > 10)
1 5 10 Outside      // False (5 is within range)
```

Equality Comparison Example

```
// Definition for not-equal comparison
( Scalar Scalar : Flag ) Def NotEqual
    =              // Check equality
    Not            // Invert result
End
```

Boolean Comarison Examples

```
5 3 NotEqual       // True  (5 != 3)
4 4 NotEqual       // False (4 != 4)

// Multiple equality comparisons
5 5 = 3 3 = And    // True  (5=5 AND 3=3)
5 5 = 3 4 = And    // False (5=5 AND 3=4)
5 6 = 3 3 = Or     // True  (5=6 OR 3=3)
5 6 = 3 4 = Or     // False (5=6 OR 3=4)
```

Complex Conditions

```
// Check if array index is valid (0 <= i < size)
( Var Var : Flag ) Def ValidIndex  // (i size -- flag)
    Over 0 >       // i > -1
    Swap Over <    // i < size
    And            // Combine conditions
End
```

Examples

```
// True  (0 is valid index when size is 10)
0 10 ValidIndex

// True  (5 is valid index when size is 10)
5 10 ValidIndex

// False (10 is not valid index when size is 10)
10 10 ValidIndex

// False (-1 is not valid index when size is 10)
-1 10 ValidIndex
```

Other Cpmparison Pattern Examples

```
// Definition for less than or equal
// (n limit -- flag)
( Var Var : Flag ) Def LessEqual
    1 +     // Increment limit by 1
    <       // Compare with adjusted value
End

// Definition for greater than or equal
// (n limit -- flag)
( Var Var : Flag ) Def GreaterEqual

    1 -     // Decrement limit by 1
    >       // Compare with adjusted value
End

// Definition for inclusive range check
// (n low high -- flag)
( Var Var Var : Flag ) Def InRange
    Rcw           // Get n to middle
    Over          // Copy low
    1 - >         // n > (low-1)  [>= low]
    Rcw Rcw       // Get high to top
    1 + <         // n < (high+1) [<= high]
    And           // Combine conditions
End
```

Examples

```
5 5 LessEqual     // True  (5 <= 5)
6 5 LessEqual     // False (6 <= 5)
4 5 LessEqual     // True  (4 <= 5)

5 5 GreaterEqual  // True  (5 >= 5)
6 5 GreaterEqual  // True  (6 >= 5)
4 5 GreaterEqual  // False (4 >= 5)

5 1 10 InRange    // True  (1 <= 5 <= 10)
1 1 10 InRange    // True  (1 <= 1 <= 10)
10 1 10 InRange   // True  (1 <= 10 <= 10)
0 1 10 InRange    // False (1 <= 0 <= 10)
11 1 10 InRange   // False (1 <= 11 <= 10)
```

Error Handling

Errors are generated by `err-code Raise`, which can be caught using a surrounding `OnError Catch` block or by referencing the system error object.

Table 34. Error Methods

Operation	Description
(:) code	Returns the current error code
(:) message	Returns the error message associated with the current error code
(:) Clear	Clears the error code

Table 35. Error Handling Words

Operation	Description
(:) Abort	Exits the interpreter printing an error message
(:) Catch	Marks the beginning of the error exception handler
(:) End	Resets the system error variable and exception handler, then resumes execution.
(:) OnError	Starts an error handling block, by setting the exception handler to point to the upcoming Catch word
(Byte :) Raise	Sets the system error variable to the value of T and raises an exception

Error Handling Example

```
// Divide by zero
( Word Word : Word ) Divide
    OnError
        /
    Catch
        // Get the error code
        // Test for divide by zero
        error.code err-div0 =
        If
            // Print Error message
            error.message Print
        Else
            Abort
        End
    End
End
```

NOTE	'err-div0' is an example identifier for the Divide by zero error code, consult the Error Message Reference for correct values

Branching

There are two types of Branch: `condition If true-actions Else false-actions End` and `value Case value match-value Of of-actions ; Else of-actions ; End`

If, Else, End

Kingfisher provides conditional branching through the `If···Else···End` construct. this structure allows programs to execute different code blocks based on boolean conditions. The If block is always executed when the condition is true, while the optional Else block is executed when the condition is false.

Table 36. If Branch Operations

Operation	Description
(Flag :) If	Continues execution if flag is true. Branches if false
(:) Else	Optional target for branch from if
(:) End	Target for the branch from If when Else is not present. End is always executed.

If Branch Examples

```
// Check if number is positive
( Var : ) Def CheckPositive
    String( pos "Positive" )
    String( neg "Negative")

    0 >
    If
        pos.Get Print
    Else
        neg.Get Print
    End
End

// Check if equal to specific value
( Var : ) Def CheckTen
    String( result "Equal to 10")

    10 =
    If
        result.Get Print
    End
End
```

If Usage Examples

```
5 CheckPositive     // Prints: Positive
-3 CheckPositive    // Prints: Non-positive
5 CheckTen          // No output
10 CheckTen         // Prints: Equal to 10
```

Case, Of, Else, End

The Case structure provides a way to match a Byte value against multiple values and execute corresponding code blocks. When a match is found, the associated block is executed. If no matches are found, the optional Else block is executed. Each case must be a value between 0-255.

Each case begins with an Of and ends with a `;`, all words in between '`Of' and `;` are executed if the `Case` value matches the `Of` value. An optional `Else` clause captures any unmatched cases.

Table 37. Case Branch Operations

Operation	Description
(Byte :) Case	Opens the `Case` and matches Number with the following `Of` clauses
(Byte :) Of	Use 1 or more 'Of'. Matches number with the `Case` value and continues if equal. Branches to next `Of` if false
(:) ;	Ends the `Of` clause and branches to `End`
(:) Else	Optional clause executed if none of the preceeding of clauses matched

Examples

```
// Days in month example
( Byte : ) Def MonthDays
    String( days31 "has 31" )
    String( days28 "has 28" )
    String( invalid "is an invalid month")

    Case
        1B Of            // January
            days31.Get Print
        ;
        2B Of            // February
            days28.Get Print
        ;
        3B Of            // March
            days31.Get Print
        ;
    Else
            " Invalid" Print
    End
End
```

Examples

```
1B MonthDays         // Prints: has 31 days
2B MonthDays         // Prints: has 28 days
13B MonthDays        // Prints: is an invalid month
```

Iterators and Range

Collections in Kingfisher provide an iterator interface that enables sequential access to their elements. this interface consists of three core methods that support iteration control and data access during loops. The iteration interface is supported by Strings, Arrays, StringArrays and Ranges. Its purpose is to control Loops.

Range Interface

A `Range` is a form of collection that supports the iteration interface only. A Range is created using the syntax; `start-count (inclusive) end-count (exclusive) step-size Range name`, where step-size is a signed value. When step-size is negative start-count must be greater than end-count

Table 38. Collection Iteration Words

Operation	Description
(: Num) First	Restarts the Iteration and returns the first value
(: Flag) IsLast	Checks if the last iteration was the final one
(: Num) Next	Gets the next value from the collection

NOTE If Next tries to iterate past the end of the collection, an error is raised. The `OnError` construct can be used to handle these errors and terminate loops

Range Iteration Examples

```
0 10 1 Range decade     // A range of 10 values 0 through 9
2 21 2 Range evenRange  // A range of 10 values 2 to 20
9 -1 -1 Range countDown // A range of 10 values from 9 through 0
```

Collection Iteration Examples

```
// Create an array of 5 numbers
Array[ array 1 2 3 4 5 ]

array.First     // return 1
array.Next      // return 2
array.Next      // return 3
array.Next      // return 4
array.IsLast    // return false
array.Next      // return 5
array.IsLast    // return true
array.Next      // Error: "End of Iteration"

// Create a range of 5 numbers
1 6 1 Range range

range.First     // return 1
range.Next      // return 2
range.Next      // return 3
range.Next      // return 4
range.IsLast    // return false
range.Next      // return 5
range.IsLast    // return true
range.Next      // Error: "End of Iteration"
```

While Loop

The `While` loop is a conditional loop with a condition at the start, in Kingfisher the loop consutruct is as follows: `Do` condition `While` actions `End`. End always returns to Do and the loop will exit at while if the condition is false. An infinite loop is therefore `Do true While actions End`.

Table 39. While Loop Words

Operation	Description
(:) Do	Marks the beginning of a `Do` loop and the target for the closing `End`
(Flag :) While	When the condition is file `While` will branch to the first instruction after end
(:) End	`End` marks the end of the `While` loop, and always branches back to `Do`

While Loop Example 1

```
// Count to 10 example
( : ) Def CountToTen
    Var count

    100 count.Set
    Do
        count.Get 0 =       // Check if end of count
    While
        1 count.Sub         // Print current count
        count.Get Print
    End

End
```

While Loop Example 2

```
 Array[ primes 2 3 5 7 11 13 ]
// Sum numbers example
( : ) Def SumToFive
    Var index

    Do
        // Check if at and of list
        index.Get primes.Length =
    While
        // Print the value
        index.Get Print

        // Increment the index
        1 index.Add
    End
    CR Print
End
```

For Loop

The `For` loop in Kingfisher is implemented as a `foreach` loop. A foreach loop iterates through a sequence of values using an iterator interface rather than using a bounded index. This approach (adopted from Python), reduces the risk of referencing values that are out of range. This construct is possible because of the iterator interface discussed previously in Iterators and Range. A traditional For loop can be constructed using `Range` which exposes an iteration interface and is compatible with For.

Table 40. For Loop Words

Operation	Description
(: Scalar) For name	Marks the beginning of the For loop, the branch target and also fetches the next iterator. When `For` has processed the last iterator it branches to the instruction following `End`
(:) End	`End` marks the end of the `For` loop, and always branches back to `For`
(:) Leave	Leave the `For` loop cleanly and exit after `End`

For Array Example

```
 Array[ primes 2 3 5 7 11 13 ]
// Print 1 to 5 example
( : ) Def PrintOneToFive
    For primes      // iterate through the primes array
        Print       // Print the first six primes
    End
    CR Print
End
```

For Range Examples

```
// Print 1 to 5 example
( : ) Def PrintOneToFive
    1 7 1 Range r

    For r           // iterate through the r range
        Dup *
        Print       // Print the first six squares
    End
    CR Print
End
```

Part III: Assembly Language and System Fundamentals

This section covers the internal structures and mechanisms used by Kingfisher for managing system types, variables, and memory. It includes details about dictionary structures, system variables, primitives, and assembly language support.

Chapter 25. Assembly Language

While system primitives provide low-level access to system state, assembly language support enables direct hardware control and optimisation. Assembly source code in Kingfisher integrates seamlessly with high-level constructs while maintaining safety constraints.

Design Philosophy

Kingfisher's assembly language implementation is guided by several core principles that align with the overall language design. The core principles of safety and correctness are paramount, it is essential that while assembly language provides direct hardware access Kingfisher's safety guarantees are maintained without introducing additional complexity. This is achieved through compile time verification of memory access, type checking of operands, and enforcement of system protection boundaries wherever possible.

Explicit intent is another key principle. Unlike traditional assemblers that infer addressing modes from operand syntax, Kingfisher requires explicit specification of addressing modes. While this creates slightly more verbose code, it eliminates ambiguity and potential errors.

Traditional vs Explicit Addressing Example

```
 ; Traditional assembly - mode inferred from syntax
  LDA #42      ; Immediate mode
  LDA $1000    ; Absolute mode

// Kingfisher - explicit addressing modes
  lda imm 42            // Immediate mode
  lda abs 0x1000        // Absolute mode
```

Integration with high-level language features is another fundamental aspect of the design. Assembly source code exists within the scope of code blocks which are similar to Kingfisher's Defs or methods, allowing seamless interaction with Kingfisher's type system, memory management, and error handling mechanisms. This integration extends to the expression system, which allows assembly operands to include full expressions that are evaluated at compile time. Expressions are enclosed in square brackets `[ ]` and can include arithmetic operations, constants, and macro expansions.

The `.` operator is used to extract individual bytes from multi-byte values, and `@` is used to calculate the position of high and low bytes in memory. The operand must be in the range 0 to 3 where 0 returns the low order byte and 3 the high order byte. This approach makes it very easy to work with addresses and works consistently regardless of byte-order.

Assembly language maintains consistency with Kingfisher's broader syntax conventions, which includes using `//` for comments, requiring explicit block structure, and following similar naming conventions. This consistency reduces cognitive overhead when moving between high-level and assembly source code.

Example Expression Integration

```
// Compile-time expression evaluation in operands
  lda imm   0 # ptr      // High byte of ptr address
  sta zp    0 @ ptr      // Store in ptr
  lda imm   1 # ptr      // High byte of ptr address
  sta zp    1 # ptr      // Store in ptr+1
```

Assembly Language Core

The core assembly language features begin with label definitions, which provide a way to name constants and addresses for use in assembly source code. Labels can be defined using literal values or expressions and serve as symbolic references throughout the enclosing Code or Method block. These labels form the foundation for writing readable and maintainable assembly source code by providing meaningful names for values and locations in memory.

Table 41. Label Word

Operation	Description
(Num :) Label name	Define a constant label that can be used in operands. Labels are only used during the assembly process and do not occupy memory at runtime

Label Definition Examples

```
// Example: Label definition and usage
( : ) labelExample Method MethodName
    Assembly

    // Define constant labels
    0xFF Label minus-one        // Define constant -1
    0x1000 Label screen         // Screen memory location
    [ 40 50 + ] Label width     // Can use arithmetic in definition

    // Demonstrate label usage in code
    lda imm minus-one           // Load 0xFF
    sta abs screen              // Store to $1000

    // Demonstrate expressions with labels
    lda abs [ screen width + ]  // Calculate address

    // Demonstrate address labels
start:                          // Address label
    lda imm 0                   // Load zero
    beq start                   // Branch to start
End Method
```

NOTE

1. Used during Assembly only.

2. Zero memory footprint
3. Label directive creates named constants
4. Can use literal values or arithmetic expressions
5. Labels can be used in operands and expressions
6. Different from regular address labels

Instruction Set Support

Instruction Set Support provides comprehensive access to the complete 65xx instruction set through Kingfisher-specific syntax adaptations. The system accommodates both standard 6502 and enhanced 65C02 instructions, ensuring full compatibility across the processor family.

The design principles and syntax conventions established here are intended to extend beyond the 65xx family. Future versions of the assembler will support other classic CPU architectures including the Intel 8086, Motorola 68000, and Zilog Z8000 processor families. This extensible approach ensures consistent syntax and programming patterns across different instruction sets while maintaining the explicit addressing modes and safety guarantees that are central to Kingfisher's design.

Instruction Set

Mnemonic	Operation	Description
adc	Add with Carry	A = A + M + C
and	Logical AND	A = A & M
asl	Arithmetic Shift Left	C ← [7:0] ← 0
bbr0-7[C02]	Branch on Bit Reset	Branch if M[bit] = 0
bbs0-7[C02]	Branch on Bit Set	Branch if M[bit] = 1
bcc	Branch if Carry Clear	Branch on C = 0
bcs	Branch if Carry Set	Branch on C = 1
beq	Branch if Equal	Branch on Z = 1
bit	Bit Test	N = M7, V = M6, Z = A & M
bmi	Branch if Minus	Branch on N = 1
bne	Branch if Not Equal	Branch on Z = 0
bpl	Branch if Plus	Branch on N = 0
bra[C02]	Branch Always	Branch regardless of flags
brk	Force Break	Push PC, Push SR
bvc	Branch if Overflow Clear	Branch on V = 0
bvs	Branch if Overflow Set	Branch on V = 1
clc	Clear Carry Flag	C = 0

Mnemonic	Operation	Description
cld	Clear Decimal Mode	D = 0
cli	Clear Interrupt Disable	I = 0
clv	Clear Overflow Flag	V = 0
cmp	Compare	A - M
cpx	Compare X Register	X - M
cpy	Compare Y Register	Y - M
dec	Decrement Memory	M = M - 1
dea[C02]	Decrement A	A = A - 1
dex	Decrement X Register	X = X - 1
dey	Decrement Y Register	Y = Y - 1
eor	Exclusive OR	A = A ^ M
inc	Increment Memory	M = M + 1
ina[C02]	Increment A	A = A + 1
inx	Increment X Register	X = X + 1
iny	Increment Y Register	Y = Y + 1
jmp	Jump	PC = Address
jsr	Jump to Subroutine	Push PC, PC = Address
lda	Load Accumulator	A = M
ldx	Load X Register	X = M
ldy	Load Y Register	Y = M
lsr	Logical Shift Right	0 → [7:0] → C
nop	No Operation	No Change
ora	Logical OR	A = A | M
pha	Push Accumulator	Push A
php	Push Processor Status	Push SR
phx[C02]	Push X Register	Push X
phy[C02]	Push Y Register	Push Y
pla	Pull Accumulator	A = Pull
plp	Pull Processor Status	SR = Pull
plx[C02]	Pull X Register	X = Pull
ply[C02]	Pull Y Register	Y = Pull
rmb0-7[C02]	Reset Memory Bit 0-7	M[bit] = 0
rol	Rotate Left	C ← [7:0] ← C

Mnemonic	Operation	Description
ror	Rotate Right	C → [7:0] → C
rti	Return from Interrupt	SR = Pull, PC = Pull
rts	Return from Subroutine	PC = Pull
sbc	Subtract with Carry	A = A - M - !C
sec	Set Carry Flag	C = 1
sed	Set Decimal Flag	D = 1
sei	Set Interrupt Disable	I = 1
smb0-7[C02]	Set Memory Bit 0-7	M[bit] = 1
sta	Store Accumulator	M = A
stp[C02]	Stop	Enter Stopped State
stx	Store X Register	M = X
sty	Store Y Register	M = Y
stz[C02]	Store Zero	M = 0
tax	Transfer A to X	X = A
tay	Transfer A to Y	Y = A
trb[C02]	Test and Reset Bits	M = M & !A
tsb[C02]	Test and Set Bits	M = M | A
tsx	Transfer SP to X	X = SP
txa	Transfer X to A	A = X
txs	Transfer X to SP	SP = X
tya	Transfer Y to A	A = Y
wai[C02]	Wait	Enter Wait State

Addressing Modes

Mode	Addr Mode	Operand	Example
Immediate	imm	value	lda imm 0xFF
Absolute	abs	address	sta abs 0x1000
Zero Page	zp	address	lda zp 0x20
Indirect	ind	address	jmp ind 0x1000
Absolute X-Indexed	abs-x	address	lda abs-x 0x1000
Absolute Y-Indexed	abs-y	address	lda abs-y 0x2000
Zero Page X-Indexed	zpx	address	lda zpx 0x20
Zero Page Y-Indexed	zpy	address	ldx zpy 0x20

Mode	Addr Mode	Operand	Example
Indirect X	ind-x	address	lda ind-x 0x20
Indirect Y[C02]	ind-y	address	lda ind-y 0x20
Zero Page Indirect[C02]	zpi	address	lda zpi 0x20
Absolute Indexed Indirect[C02]	abs-ind	address	jmp abs-ind

Kingfisher Assembly Example

```
    // Define base address for operation
    0x1000 Label test-base      // Starting address for clear

    // Initialize counter
    ldy imm 0x00               // Set Y register to zero

    // Clear memory loop would follow...
```

NOTE

- [C02] indicates 65C02-specific instructions and addressing modes
- All addressing modes must be explicitly specified
- Immediate values can use calculation syntax: lda imm [2 4 +]
- Branch instructions use absolute addressing syntax for consistency
- Zero page addresses must be within 0x00-0xFF range

Assembly Directives

Assembly Directives control the assembly process and data definition while managing symbol definitions and memory layout. The system includes support for executing Kingfisher words during the assembly process, providing essential control.

*Table 42. Data Definition Directives**

Directive	Description
byte value value 'A'	Define byte(s) from numeric values or character literals
byte "text"	Define counted string (length byte followed by characters)
word value value	Define word(s) (16-bit)
long value value	Define long(s) (32-bit)
align boundary	Align to power of 2 boundary
space count	Reserve count bytes of space
fill count value	Fill count bytes with value
[	Execute the Kingfisher interpreter and return results to the assembler.

Directive	Description
]	Switch back to Assembly Language.
(Num Byte : Byte) #	Returns byte T from the multi-byte word N
(Num Byte : Byte) @	Returns the byte @ address N + T

Table 43. Section Control

Directive	Description
code	Optional: Begin code section (65xx uses unified address space)
data	Optional: Begin data section (65xx uses unified address space)
z-page	Begin zero page section. Address is required for system variables, otherwise space is auto-allocated

Assembler

The assembler in Kingfisher provides a structured framework for creating executable machine code by extending the capabilities of the compiler. The assembly process uses the interpreter to find opcodes and directives in the dictionary and the executes them to compile them into the source, in much the same way as higher level source is compiled. This ability enables switching between compiler and assembler easily, allowing for integration and access to Kingfisher's vocabulary.

There are two mechanisms to access assembly language. The first is the ability to create a code definition in similar way to `Def`. The second is to enable assembly in a linked method.

Table 44. Assembly Language Words

Operation	Description
(:) Assembly	Switches to assembly language within a method definition
(in : out) Code name	Creates a new definition and enables the Assembler
End	Closes the definition and switched back to the interpreter

Code Block Example

```
( Byte Byte -- Byte ) Code Add
    lda zp-x 0 @ [ stack 1 + ]   // Load first parameter
    clc                         // Clear carry
    adc zp-x 0 @ [ stack 1 + ]   // Add second parameter
    lda imm  0                  // high byte is zero
    sta zp-x 1 @ [ stack 1 + ]   // Store result
    inx                         // Adjust stack
    rts
End
```

Assembly Method Example

```
( Byte Byte -- Byte ) type-name Method
    Assembly
    //
    lda zp-x 0 @ [ stack 1 + ]   // Load first parameter
    clc                         // Clear carry
    adc zp-x 0 @ [ stack 1 + ]   // Add second parameter
    lda imm 0                   // high byte is zero
    sta zp-x 1 @ [ stack 1 + ]   // Store result
    inx                         // Adjust stack
    rts
End
```

Macro Processing

Macro processors can be used in compilers or assemblers to abbreviates language syntax, and makes it easier to repeat complex construct consistently. Macros can also have parameters enabling programmers to create syntactic patterns that can be reused in different contexts.

The Kingfisher macro processor is integrated into the compiler and assembler enabling programmers to use macros in either system.

Table 45. Macro Words

Operations	Description
(:) Macro name	Defines a new macro called name.
(%	Starts a macro parameter that contains spaces and other delimiters and operators
%)	Ends a macro parameter and stores it in the word definition
$ name macro-name	Passes name to macro-name
End	Closes the macro, switches off the macro processor and back to the Interpreter

Macro High Level Example

```
// duplicate top two stack items
(: ) Macro Dup2
    Over Over
End

3 5 Dup2        // Results in 3 5 3 5
```

Macro Assembly Example

```
(: ) Macro SetPtr
    lda imm     $1 0 #          // load A with low byte of parameter
    sta zp      ptr1 0 @        // store A in pointer low byte
    lda imm 1   $1 1 #          // load A with high byte of parameter
    sta sp 1    ptr1 1 @        // store A pointer high byte
End

( Byte : ) Code StoreAtPointer
    [ addr SetPtr ]           // Pass the string addr to the macro
    // [ (% addr %) SetPtr ]    // Alternate form

    0x20 Label ptr1

    // Macro expands to
    lda imm     addr 0 #
    sta zp      ptr1 0 @
    lda zp      addr 1 #
    sta imm     ptr1 1 @

    lda zp-x    [ stack 1 + ]   // load a with value on stack
    inx                         // drop value on stack
    sta ind-y   ptr1            // store in memory
    rts
End
```

Chapter 26. System Data Structures

Additional controls are required to manage system variables, which includes more constructors and methods to give the variable a fixed address, and to enable creation of singletons.

Singletons are declared using `Static` which uses a similar syntax to 'Type'. Unlike standard types, static types are constructed when they are defined, which limits the number of instances to one. A static instance is accessed using linked methods just like any other type.

All data structures support the `Locate` method which removes a variable from the heap and locates it at a specific location.

The `Register` word is used to mark the variable as a system register and the word is allocated a position in the system register space. If locate is used with a register the address must be within the system register address space.

Locate and Register words must immediately follow a constructor or static definition to avoid memory fragmentation.

Table 46. System Data Constructors

Constructor	Description
(:) Static name	Creates a new static type

Table 47. System Data Words

Operations	Description
(Var :) Locate name	The locate method must be used immediately after variable creation and before any other defining words are used. The named variable is moved from the heap and located at the specified address
(:) Register name	The register method must be used immediately after variable creation and before any other defining words are used. The named variable is moved from the heap into the system register page.

Static Type Definitions

Static types in Kingfisher are defined using a structured template syntax that enables precise control over memory layout and field access. The static definition begins with a stack signature followed by the 'Static' keyword and a list of field definitions, terminated with End. For a detailed description of type methods, and constructors see Type Definitions.

Static Type Definition Examples

```
// Example type with explicit addressing
( : ) Static chain
    1 Var : activated
    1 Var : extended
    1 Var : restricted
    1 Var : excluded
End

Register chain      // Chains are a register
0x010 Locate chain  // ... fix them at 0x010


// Example type with auto-increment addressing
( : ) chain Method Activate
    // code to activate activated chain
End
```

IMPORTANT	Static types trade flexibility for efficiency by allocating memory at compile time and eliminating constructor overhead. Regular types provide instance isolation but require runtime construction.

Chapter 27. Interrupts

Interrupt handling in Kingfisher provides a mechanism for responding to external events in a timely and efficient. While powerful, it's crucial to use interrupts carefully to avoid performance degradation and system instability.

Interrupt Handling System

The Kingfisher interrupt handling system comprises three key components, as shown in the following table.

System Interrupt Handler	The core mechanism that receives and interrupts from the CPU.
Interrupt Prioritisation	A system for assigning priorities to different interrupt sources, ensuring that critical events are handled promptly. Interrupt prioritisation ensures that higher priority interrupts are handled before lower priority interrupts, preventing critical events from being delayed.
Interrupt Service Routines	User-defined functions that are executed in response to specific interrupts. For example an ISR might handle a timer interrupt by updating a system clock, or a keyboard interrupt by reading keypresses from the input buffer.

Interrupt Processing

Interrupt processing leverages the CPUs built-in interrupt mechanism. When an interrupt occurs, the actions listed in the following table are performed.

Interrupt Acknowledgement	The CPU signals an interrupt to the Kingfisher system, which responds to the interrupt at the end of the current CPU instruction and the CPU registers are saved on the system stack.
Customer Increment	The system interrupt handler increments an internal counter. The CPU registers are restored and execution continues.

Deferred Handling	The actual processing of interrupts is deferred until the end of the current definition. The deferred approach minimises the need for extensive state preservation within the interrupt handler itself. By relying on the existing stack-based parameter passing mechanism of Kingfisher, the overhead of handling interrupts is kept to a minimum. The inner interpreter checks the interrupt counter at the end of each definition (next), and calls the interrupt processing routine only when the counter is greater than zero.
Interrupt Processing	The system proceeds to execute the appropriate interrupt service routines based on their priorities.
Counter Decrement	The interrupt counter decrements following completion of the interrupt processing routine. This mechanism ensures that interrupts are handled efficiently while minimising disruptions to the normal flow of program execution.

NOTE This approach relies on the interrupt frequency being lower than the overall interrupt handling period. It is the responsibility of the programmer to ensure the interrupt frequency and handling period do not impact the performance of the application.

Interrupt Management

Interrupts are managed using a singleton object called `isr`. The following tables lists all the methods linked with the `isr` object.

Table 48. ISR Methods

Operations	Descriptions
(Byte :) AddHandler name	The word specified by name is added to the specified interrupt priority as the ISR
(Byte :) Disable	Disable interrupts at teh specified priority level only. Priority 0 is all interrupts, while 1-4 are individual interrupt priorities with 1 being the highest and 4 lowest
(Enable :) Enable	Enable interrupts at the specified priority level only

Interrupt Example

```
// define a counter in register space
Byte Counter
counter.Register

// define constants tp represent interrupt priorities
0 Constant irq-all  // Refers to all interrupts
1 Constant irq-p1   // Highest Priority
2 Constant irq-p1
3 Constant irq-p1
4 Constant irq-p1   // Lowest Priority

// Word to count 100 x 10mS events
// Counter will roll over every 1 second
( : ) Def Count
    counter.get 100 =
    If
        counter.Reset
    Else
        1 counter.Add
    End
End

// Initialis counters and interrupts
( : ) Def InitCounter
    // Disable all interrupts before initialisation
    irq-p4 isr.Disable

    // Reset the counter to zero
    counter.Reset

    // Initialise timer device to 10mS

    // Add the Count word as the priority 4 interrupt handler
    irq-p4 isr.Addhandler Count

    // Enable priority 4 interrupts
    irq-p4 isr.Enable
End
```

Part IV: Runtime Features

Runtime features provide the essential services needed by programs during execution, and provide file access, system primitives and memory management services. While many modern systems hide such features behind layers of abstraction, Kingfisher uses a more traditional direct approach that is better suited to vintage hardware.

The runtime features focus on basic safety and reliability, providing simple type checking and clear error reporting while maintaining direct access to system resources. This helps catch common programming errors at compile-time while still allowing efficient use of the limited resources available on vintage systems.

Chapter 28. Introduction

The runtime system provides an effective programming environment and execution environment while maintaining efficient operation on vintage hardware. It bridges the gap between high-level programming concepts and the underlying system capabilities through a set of carefully designed abstractions.

The runtime system is built on four core abstractions, which are: a uniform file system interface providing access to various storage devices, an object system supporting static type creation and reference management, a Memory management controlled by scope, and a simple error handling system providing reliability.

Each subsystem is designed to balance type safety and an improved programming system with vintage system constraints.

The file system abstraction, for example, provides familiar operations while accommodating the specific characteristics of vintage storage devices. Similarly, the object system implements modern programming concepts while maintaining efficient memory use and predictable performance characteristics.

The following sections detail these runtime features, providing both vocabulary and practical usage examples. The abstractions hide much of the underlying complexity and provide increased safety through type checking.

Chapter 29. I/O Subsystem

I/O operations in Kingfisher are organised into two main categories: Console I/O and File I/O. The Console I/O module provides the foundation for Kingfisher's Print functionality, along with input handling through Read, and IsReady operations. These console operations abstract the underlying system-specific details while maintaining full access to the hardware capabilities. File I/O handles all aspects of file and device access, building upon the host system's native file handling capabilities.

Console I/O

Input and output operations form the foundation of any programming language's interaction with the outside world. Standard abstractions provide console operations with input, output and error streams. Streams provide buffering and error handling features, simplifying the task of I/O operations.

Three default streams: in, out and err are used to read user input data, output user data to and to notify user of any errors. Usually these streams connect to the system's standard console devices, although they can be redirected to other devices or files as needed.

The std-io module provides the declarations that establish these default streams, which connect to the system's standard console devices by default, but support redirection as needed.

Stream Definitions

Additional streams can be defined as required using the [Stream Constructors] from the table below.

Initially the system binds the Input streams to the keyboard device while output streams connect to the screen device. These default bindings can be modified through the stream redirection methods described in subsequent sections.

Table 49. Stream Constructors

Constructor	Description
(:) InputStream name	Creates an input stream connected to the default input device
(:) OutputStream name	Creates an output stream connected to the default output device

Console I/O Example

```
Module std-io
    // Define the default console input stream
    InputStream in

    // Define the default Console output stream
    OutputStream out

    // Define the default Console error stream
    OutputStream err
End
```

Stream Redirection

Kingfisher's stream architecture allows programs to dynamically redirect input and output operations to different devices or files. This flexibility enables programs to read from or write to various system resources without modifying the underlying code. Stream redirection is accomplished through the Bind method, which accepts a device or file specification as a string parameter.

The `<Input|Output>Stream.Bind` method provides a uniform interface across different system devices. By accepting string-based device specifications, the language remains independent of system-specific device naming conventions while still providing full access to the underlying hardware.

Table 50. Stream Methods

Operation	Description
(String :) Bind	Binds a stream to the specified device

Stream Redirection Examples

```
// redirect output to a file
"readme" out.Bind         // redirect output to a file
"TAP:output" out.Bind     // redirect output to Tape

// redirect output to a specific disk, drive and file
"DSK1:0:output" out.Bind
"PRN1:" out.Bind          // redirect to a printer
"COM1:" out.Bind          // redirect all IO to COM1:
"COM1:" in.Bind

"SCN:" out.Bind           // reset out to SCN:
"KBD:" in.Bind            // reset in to KBD:
```

Stream IO Operations

The core functionality of streams is implemented through three fundamental operations: Read,

Write, and IsReady. These methods provide byte-level IO operations that form the foundation for all stream-based data transfer. The IsReady operation allows programs to check input availability without blocking, while Read and Write operations handle the actual data transfer.

In addition to the these core operations there are functions to read and write a line of characters terminated with a new-line (0xA)character, as well as arguments to get and set position of the cursor in the stream buffer.

Table 51. Stream IO Methods

Method	Description
(Long :) GetPos	Gets the position of the stream pointer, and returns it as a long. This function is used to get the position in the stream which may not be defined for all devices. Unsupported devices always return 0.
(: Flag) IsReady	Checks to see if the input stream is ready
(: Byte) Read	Reads a value from the InputStream
(: Byte) ReadLn	Reads values from the InputStream until the end of line is reached
(Long :) SetPos	Positions the stream pointer. This function is used to set the position in the output stream which may not be defined for all devices. Unsupported devices perform a no operation.
(Byte :) Write	Writes a value to the OutputStream
(String :) WriteLn	Writes a string to the output stream terminated with a new line

NOTE For screen operation the position represents the address from the top left corner of the screen, which is equivalent to `row-number line-length * col-number +`.

Stream IO example

```
InputStream in
OutputStream out

( | ) EchoServer
Do
    True
While
    in.IsReady
    If
        in.Read out.Write
    End
End
```

Stream Error Handling

Stream operations can encounter various hardware and timing-related issues during device interactions. The stream system provides automatic error recovery mechanisms and supports explicit error handling through OnError statements. This ensures programs can gracefully handle device failures and maintain reliable IO operations.

When redirecting streams, three primary errors may occur:

1. **Device Not Present:** The specified device is not connected or available
2. **Device Timeout** The device failed to respond within the expected time
3. **Buffer Overflow:** The stream received more information than it was able to process

All errors will automatically reset the stream to its default device (KBD: for input, SCN: for output) unless handled by an OnError recovery statement.

Examples

```
// Example with error handling
OnError
    " PRN1:" out.Bind
    // ... device operations ...
Recovery
    // Handle printer offline/not present
    " Device error - reverting to screen" Print
End

// Example without error handling - will auto-reset to SCN:
// If printer not present, reverts to screen
" PRN1:" out.Bind
```

File I/O

Filesystem operations on vintage systems such as the Commodore machines differ significantly from modern Unix or Windows systems. These early computers had minimal abstraction layers, requiring users to communicate directly with disk drive controllers through low-level commands. This made file operations complex and system-specific.

Kingfisher addresses this complexity through its file-io module, which provides a modern abstraction layer for file operations. This approach allows programmers to work with files using familiar patterns similar to modern languages like Python, while the system handles the underlying vintage hardware interactions transparently.

The file-io module organises storage operations into three main concepts, which are the storage devices for accessing storage medai, volumes for disk operations, and files for individual file management. This separation provides a hierarchy for managing multiple devices, volumes and file level operations.

Storage Devices

Kingfisher supports three types of storage device, each with their own type of file system. An abstraction layer provides a common interface to all three device types. The drive and filesystem characteristics are described in more detail in the following table.

Device Type	Description
SC1-2	Up to two SD Card interfaces using an SPI interface 8 x 1GB partitions 512 files per partition Max file size = 1GB
SD1-8	Up to Commodore compatible serial or parallel disk drives. Abstract interface removes requirement to deal with Commodore drive interface directly. No secondary addressing required
SE1-8	8 x 128/256KB Serial EEPROM devices using an I2C interface. Max possible device size is 1MB File size limited by device size 30 files max 128KB 62 files max 256KB (128KB each, hardware-linked I2C addresses)

Device IDs and File Names

Storage devices in Kingfisher use a standardised naming convention that provides consistent access across different device types. The system employs an addressing scheme to identify specific devices, drives and partitions within the storage system.

Device identifiers follow a structured format combining device type (SD, SC, or SE) and unit number. The system uses a colon delimiter to separate device, drive and partition specifications. For example, "SD1:0" references drive zero on Commodore device SD1, whilst "SC1:7" indicates partition 7 on SD-Card number 1.

File naming follows established conventions whilst accommodating the specific requirements of vintage systems. Names may contain up to 16 characters for standard devices, extending to 29 characters on SD Card systems. The system reserves certain characters for special purposes within the file system. The system also provides wildcard characters for flexible file matching - the asterisk matches any sequence of characters from its position to end of name, whilst the question mark matches exactly one character at the corresponding position.

Device identifiers can include volume names using the forward slash (/) separator. For example, "home/readme" references a file called "readme" on the volume named "home". When no volume is specified, the system uses the default volume, named "home".

Table 52. Special Characters

Character	Purpose
/	Separates volume names from file names

Character	Purpose
:	Separates device identifiers from file specifications
*	Matches any sequence of characters from its position to end of name
?	Matches exactly one character at the corresponding position

NOTE File names must not contain the special characters listed in the table above, as these are reserved for system use. Using these characters in file names will result in invalid file specifications.

Device and File Name Examples

```
"SD1:0:program"     // Device SD1, drive 0, file "program"
"home/readme"       //
"prog????.dat"      // Matches "prog" plus any 4 chars + ".dat"
"test*"             // Matches all files beginning with "test", from th default volume
```

File System and Volume Management

Access to file systems in Kingfisher is provided through volume objects. The volume object abstracts device IDs, allowing users to provide more meaningful names for the intended use. A default volume called "home" is mounted during boot.

Volumes provide disk-level operations for managing storage devices. These operations handle device initialisation, status monitoring, and directory management. Each volume represents a filesystem on a physical storage device in the system.

Before files can be created or accessed, a volume must be mounted and initialised. This abstraction allows programs to work with different storage devices uniformly, whether they are physical drives or other storage media.

Table 53. File System Words

Operation	Description
(String :) Mount name	Mount the device specified in string to the mount point called name
(:) Unmount name	Disconnect the named volume. The volume does not get deleted but stays in the unmounted state until a device is mounted
(:) Volumes	List all the available volumes and their status to the output stream
(:) IsReady name	Check if named Volume is mounted and accessible

Operation	Description
(:) Info name	Get extended information for the named volume: Device ID Filesystem capacity Filesystem Usage
(String String :) Copy	Copy file-path1 in N to file-path2 in T
(String :) Create	Create a new file using the specified name
(String :) Delete	Delete the specified file
(String1 String2 :) Rename	Changes the name of the file specified in N to the name specified in T
([String] :) List	List all the files, applying the optional specified filter

File Operations

File objects provide the primary interface for reading and writing data to storage devices. These operations manage individual files within a volume, handling tasks such as creation, deletion, and modification of file content.

Files in Kingfisher build upon the volume abstraction layer to provide consistent access across different storage systems. Each file object maintains its own state including mode, position, and associated volume, allowing multiple files to be handled simultaneously.

The file system implements standard operations like open, close, read, and write while managing the complexities of vintage hardware interactions transparently. This approach enables familiar file handling patterns while ensuring reliable operation on classic hardware.

Table 54. File Constructor

Operation	Description
(String :) File name	Create a file object called name using the file path specified in string

Table 55. File Methods

Operation	Description
(:) Close	Close the file
(String Byte :) Open	Open the file name specified in N using the specified mode T
(ByteArray : Var) Read	Read max of array length into array T. Returns number of bytes read
(String :) ReadLn	Read max of string length into string T or until new line

Operation	Description
(ByteArray Var :) Write	Write max of array length or T bytes from byte array
(String :) WriteLn	Write string length bytes from string T followed by a newline

NOTE Seek operations are not supported. On Commodore devices only USR file types are supported. Other file types should be converted to USR before using with Kingfisher, this way there is no confusion between Commodore and Aves files

File Modes

File modes control how Kingfisher interacts with files during read and write operations. Each mode provides specific access patterns that determine whether a file can be read, written, or appended to. The mode must be specified when opening a file and remains fixed until the file is closed.

Table 56. File Mode Modifiers

Mode	Description
'a'	Append mode - Opens the file for adding data to the end
'c'	Create mode - Creates a new file only. Returns an error if the file already exists. This is the safest mode for creating new files as it prevents accidental overwrites.
'r'	Read mode - Opens an existing file for reading
'w'	Write mode - Opens an existing file for writing. WARNING: Due to the infamous "save-with-replace" bug in Commodore DOS firmware, this mode may corrupt existing files on certain drives. To avoid this Kingfisher does not use or support the CBM DOS "@filename" syntax. Write mode instead uses rename, create delete to achieve the same thing safely.

File Examples

```
// Mounts two drives on separate volumes
"DSK1:0" root.mount vol1              // Drive zero of disk 1

// Create a file, file must not exist
( : ) Def WrHello
    "vol1/hello.kf" 'c' Open f1     // Create a file object

    // open file for create, must not exist
    "Hello, world!" f1.WriteLn        // Write message
    f1.Close
End

// Read a file
( : ) Def RdHello
    10 String msg                   //

    // open file for create, must not exist
    "hello.kf" 'r' Open f1          // open hello.kf for read
    msg f1.ReadLn                   // Read message into string
    f1.Close
End
```

Operating System Management

Kingfisher provides a comprehensive set of operating system management operations through the `os` module. These operations enable file and volume management similar to contemporary operating systems, with commands optimised for command-line usage.

The operations support essential file system tasks including copying, creating, deleting, formatting, renaming and validating files and volumes. Each operation is designed to work consistently across different storage devices.

Table 57. Operating System Words

Operation	Description
(:) Copy(<file-path1> <file-path2>)	Copy file-path1 to file-path2
(:) Mount(name "device-id")	Mount the device-id and create a volume called name
(:) New(<file-path>)	Create a new file called file-path
(:) Delete(<file-path>)	Delete the file called file-path
(:) Format(<device-id> <media-name>)	Format the volume ready for use
(:) Rename(<file-path1> <file-path2>)	Changes file-path1 to file-path2
(:) Validate(<device-id>)	Checks the media for errors, and ensures that all used blocks are marked correctly

NOTE	Parameters in angled indicate that strings without spaces can be used without using quoutes

Operating System Example

```
// Format a device
Format( se1 )
Validate ( se1 )

// Create and copy files
New( readme.txt )
Copy( readme.txt readme.bak )
Rename( readme.bak readme.bk1 )
List( *.prg )
```

Chapter 30. System Primitives

System primitives provide a means to access and manage fundamental system resources including Dictionary, and Heap. These data structures are discussed in more detail in the [Fundamental System Structure] section in Part VI.

System primitives form the foundation for extending and customising the Kingfisher compiler. They provide essential capabilities for implementing new word types and managing system resources. Developers can control memory allocation and manipulation through system primitives, while maintaining safe boundaries between different parts of the system. These primitives also enable manipulation of the system state, making them crucial for low-level system operations and compiler extensions, without accessing the resources directly.

The Compiling primitive allows words to determine whether they are being executed during compilation or runtime. This enables conditional behavior based on the compilation state, which is essential for implementing compiler directives and runtime checks.

NOTE	System primitives have access to critical system resources. Their use requires careful consideration of system stability. Incorrect usage can compromise system integrity.

Table 58. Fundamental System Words

Operation	Description
(Pointer : Var) !Addr	Tells the Compiler to treat the ptr Addr as a Var type. The size of addr must be the same as Var. This word has no effect on runtime code or code generation.
(Pointer : Var) !AsVar	Is used during compilation to tell the compiler to use a Var to represent an address type. The size of addr must be the same as Var. This word has no effect on runtime code or code generation.
(Var Var Var :) !Clone	Copies the byte values from the source address B to destination N, length T
(:) !Compile	Switch to compilation mode
(: Flag) !Compiling	Returns true if the system is currently compiling a definition, false if executing
(:) !Create name	Allocates dictionary space and creates a new entry with the specified name and signature
(String String : Var Var) !Find	Search all scope chains to find the word that matches the name in N and signature in T and return the Vocabulary's and name word's field addresses
(:) !Forward name	Adds name to forward reference list during compilation
(Var : Scalar) !Get<Scalar>	Returns the scalar value from address in T
(:) !Interpret	Switch to interpretation mode

Operation	Description
(:) !Recurse	Compiles a call to the word currently being defined
(String String : Var) !Ref	Search the Vocabulary in the active chain only for the word that matches the name in N and signature in T and return the name field address
(:) !Resolve name	Resolves pending forward references for named word
(Scalar Var :) !Set<Scalar>	Stores the scalar value N at address T

System Primitive Examples

```
// Example of compile-time vs runtime behavior
( : ) Def CompileTimeCheck
    Compiling If
        "Compiling..." Print    // Only prints during compilation
    Else
        "Running..." Print      // Only prints during execution
    End
End
```

Compiler Bootstrap Example

```
// Bootstrap compiler definition example
( String : ) !Create Def
    !Compile
    Forward End   // Add End to forward references

( : ) Def End
    !Interpret
    Recurse       // Handle nested definitions

Resolve End      // Resolve the forward reference to End
```

Compile-Time Checking Examples

```
// Example of compile-time only operation
( : ) Def CompileTimeOnly
    Compiling Not If
        "This word can only be used during compilation" Error
    End
    // ... compilation specific code ...
End

// Example of Runtime Clone operation
( Var Var : ) Def SafeClone
    // Runtime copy operation
    Dup Size Clone
End
```

Chapter 31. Memory Management

Memory management in Kingfisher operates through two complementary mechanisms that work together to provide flexible and efficient memory utilisation during program execution. The dictionary system manages program definitions and code storage, whilst the heap system handles dynamic runtime allocation needs.

When creating new definitions, the dictionary system automatically manages memory allocation, growing as needed to accommodate new entries. This automatic management ensures efficient use of memory for program structures, though direct control remains available for advanced operations. The dictionary grows upward from low memory, maintaining a linked structure that allows for both addition and removal of definitions during program execution.

The heap system provides complementary functionality for dynamic memory allocation during program execution. Operating from high memory downward, the heap allows programs to request and release memory blocks as needed. This dynamic allocation proves particularly useful when working with data structures whose size cannot be determined at definition time, or when temporary working space is required.

Dictionary Operations

Dictionary memory allocation occurs primarily through the creation and removal of definitions. The system manages the memory automatically, though direct control remains available through specific operations.

Table 59. Dictionary Words

Operation	Description
(String :) !Create name	Allocates dictionary space and creates a new entry with the specified name and signature
(String : Var Var) !Find name	Search all scope chains to find the word that matches the name and signature and return the Vocabulary name field address and name field address
(String : Var) !Ref name	Search the Vocabulary in the active chain only for the word that matches the name and signature and return the name field address
(String : Var) !Forget name	Removes the named definition and all subsequent entries from the dictionary, releasing dictionary space

Heap Operations

The heap system provides direct control over memory allocation through two primary operations. These operations allow for both allocation and deallocation of memory blocks as needed during program execution.

Table 60. Heap Management Words

Operation	Description
(Var : Var) !Allot	Allocates the specified number of bytes and returns the base address of the new memory block
(Var :) !Free	Releases all memory allocated after the specified address

Memory Management Example

```
( : ) Def ArrayExample
    100 !Allot     // Request 100 bytes of heap memory
    Dup            // Keep a copy of the base address
    50 + !Free     // Free memory from base+50 upward
End
```

Part V: Kingfisher Development

Vintage systems, like the ones Kingfisher is targeted on, often have limited memory and processing power. This restricts the types of IDEs that can be effectively used. Traditional, resource-intensive IDEs like Visual Studio or Eclipse are not suitable for such environments.

Kingfisher's architecture, inspired by Forth, encourages a functional programming approach with small, reusable functions. This aligns well with the concept of pure functions, where the output is solely determined by the input, making code more predictable and easier to maintain.

The fact that Kingfisher is designed to run on systems with limited resources, such as 6502-based computers, makes performance a primary consideration, and the development tools reflect this. To address these constraints and leverage Kingfisher's strengths, Kingfisher provides a lightweight IDE built around a lightweight text editor called Talon, which includes built-in access to the REPL and automatic retention of the last three versions. Talon's design philosophy is to provide the essential tools for coding, testing, and debugging without sacrificing performance; therefore, it prioritises efficiency and responsiveness over rich visual features. For 16-bit systems or when running on emulators with sufficient resources, future versions of Talon may include optional features like token colouring, which can be enabled or disabled based on the target system's capabilities and the user's preferences. These systems typically have more processing power and memory, making such features more practical.

Despite its minimalist design, Talon provides essential features for Kingfisher development, including efficient text editing with support for common vintage computer keyboard layouts and character sets, standard keyboard shortcuts, seamless integration with the Kingfisher REPL for interactive testing and debugging, and efficient CLI access for compilation, execution, and version control.

These features, combined with Kingfisher's focus on small, testable functions, enable developers to write and debug code quickly and effectively, even on limited hardware.

Table 61. Example Workflow

Write Code	Write Kingfisher code in the Talon text editor, focusing on small, reusable functions. Adhere to Kingfisher's functional programming paradigm.
Compile and Run	Use the Kingfisher CLI command `Load( "my-program.kf" )` to load and compile the code. Then, execute the compiled program with the `Run` command.
Test and Debug	Use the integrated REPL to interactively test individual definitions. Type in a Kingfisher expression and press Enter to evaluate it immediately. This allows for rapid prototyping and testing of small code units. The REPL provides immediate feedback, allowing you to quickly identify and correct errors.
Refactor and Optimise	Continuously refactor the code to improve its efficiency and maintainability. Use the REPL to quickly test the impact of changes.

Remote Development	For target systems with limited resources, a remote development workflow is supported. Code is written and compiled locally, then deployed to the target system using the XMODEM file transfer mechanism built into the Serin terminal emulator. XMODEM is often used for its simplicity and wide availability on embedded systems, allowing development to occur on a more powerful machine while still targeting the specific hardware.

Chapter 32. The Software Development Lifecycle (SDLC)

The software development lifecycle has six phases as shown in the [The Six Phases of the SDLC] diagram. While the initial Plan and Design phases are not directly addressed by Kingfisher itself, the remaining four phases; Implement, Test, Deploy, and Maintain are fully supported and form the core of the Kingfisher development workflow.

The Six Phases of the SDLC

1. Plan → 2. Design → **3. Implement** → **4. Test** → **5. Deploy** → **6. Maintain**

Development Environment

The Implementation phase of the SDLC is where Kingfisher's development environment plays a crucial role. Vintage systems, like those Kingfisher targets, often have limited memory and processing power. This restricts the types of IDEs that can be effectively used. Traditional, resource-intensive IDEs like Visual Studio or Eclipse are not suitable for such environments.

Kingfisher's architecture, inspired by Forth, encourages a functional programming approach with small, reusable functions. This aligns well with the concept of pure functions, where the output is solely determined by the input, making code more predictable and easier to maintain.

To address these constraints and leverage Kingfisher's strengths, Kingfisher provides a lightweight IDE built around a streamlined text editor called Talon, which includes built-in access to the Kingfisher REPL and automatic retention of the last three versions.

Implementation

Talon, the lightweight text editor, is the primary tool for the Implementation phase. Its minimalist design ensures efficiency on resource-constrained systems, allowing developers to focus on writing Kingfisher code without the overhead of a complex IDE.

Testing

The integrated REPL is essential for the Test phase. It allows developers to interactively test individual functions and code snippets, providing immediate feedback and enabling rapid prototyping. This interactive testing is crucial for verifying the correctness of Kingfisher code.

Deployment

The Deploy phase is facilitated by the Kingfisher CLI, providing tools for compiling and deploying code to the target hardware. For systems with limited resources, the remote development workflow, using XMODEM for file transfer, is particularly valuable.

Maintenance

Kingfisher's focus on small, reusable functions and its functional programming paradigm

significantly aids the Maintain phase. The modularity and predictability of the code make it easier to understand, modify, and debug. The REPL can also be invaluable during maintenance for quickly testing bug fixes or new features.

Talon IDE

The Talon Integrated Development Environment **IDE** provides a lightweight yet powerful development environment specifically designed for Kingfisher systems. It balances essential features with efficient operation on vintage hardware, offering developers a complete toolkit for writing and debugging Kingfisher applications.

The environment operates across three distinct modes, allowing developers to seamlessly transition between coding, debugging, and system operations. Each mode preserves screen real estate while providing necessary functionality for its specific purpose.

Table 62. Core Features and Specifications

Feature	Description
Display Support	Flexible width support (30, 40, 50, 80, or 100 columns)
Editor Functions	Full-screen editing with horizontal and vertical scrolling
REPL Integration	Direct access to Kingfisher REPL environment
Version Control	Automatic retention of last three file versions
Status Display	Real-time line and column position information
Screen Layout	Configurable split-screen modes for code and debug views

Table 63. Operating Modes

Mode	Function
Edit	Primary development interface providing 16-21 rows for code entry and modification
Debug	Split-screen view with 3-7 rows dedicated to debugging and REPL interaction
Menu	Overlay system providing access to file, edit, and system functions

Usage

Working with Talon follows a straightforward workflow that guides developers through the coding and testing cycle. The editor’s modal design ensures that common development tasks remain accessible whilst maximising the available screen space for code editing. A typical development session progresses through editing, testing, and file management operations as shown in the following steps.

1. Start Talon and enter Edit mode
2. Write or load Kingfisher code

3. Access the menu system using the menu key
4. Use Debug mode for testing and REPL interaction
5. Save work frequently (File menu)

NOTE	For complete reference information including menu structure, key mappings, and configuration options, see Talon IDE Reference in Part VI.

Example Application

To demonstrate Kingfisher's development capabilities, a basic terminal emulator design has been chosen. A terminal emulator is relevant to vintage computers, as many systems from that era supported multiple asynchronous terminals and multiple users. With the advent of microprocessors, these same terminals were used to debug microprocessor-based systems. The basic dumb terminal performs two primary functions: it reads characters from an asynchronous serial port and displays them, and it reads characters from the keyboard and sends them to the serial port. The first version of the application will accomplish these tasks before being enhanced to support control characters and, eventually, a subset of ANSI terminal control sequences.

```
// Include the module supporting the serial, keyboard and screen devices
include std-io

Use std-io              // Add it to the extended scope

InputStream comIn       // Now create serial port comms. channels
OutputStream comOut

// Test if character received
( : Flag ) Def GotCom
    com.IsReady
End

// Read a byte from the COM port
( : Byte ) Def RdCom
    comIn.Get
End

// Write a byte to the COM port
( Byte : ) Def WrCom
    comOut.Set
End

// Test if key available
( : Flag ) Def GotKey
    in.IsReady
End

// Read a byte from in
( : Byte ) Def RdIn
    in.Read
End

// Write a byte to out
( Byte : ) Def WrOut
    out.Print
End
```

Main Loop

```
03 Constant break-key   // break key is ^C or Stop on C64

// TtyLoop
( : Byte ) Def TtyLoop

Do
    GotKey
    If                                  // If there is a keypress to process
        RdIn Dup break-key = Not        // Check if the user has pressed the break key
        If
            WrCom                       // Send it out of the com port
            true                        // Do not exit loop
        Else
            Drop                        // Get rid of break key
            false                       // Terminate the loop
        End
    Else
        true                            // Continue rest of loop
    End
While
    GotCom                              // Is there character from the Com port?
    If
        RdCom WrOut                     // get the character and write to display
    End
End
```

Debugging the Example

Kingfisher's stack-based architecture and focus on small, testable functions make it particularly well-suited for this type of unit testing. Once the example has been compiled and saved as `example-1.kf` using the [Talon Editor] its ready to be tested. Before testing, connect a serial port from your host machine to the serial port on the Aves Nest expansion card. To test each of the words, GotCom, RdCom, WrCom, GotKey RdKey and WriteKey can be tested in the command line as shown in the following example.

Debug Example

```
" example-1.kf" Load

// Start by testing the keyboard
GotKey          // Returns False
// Press a key
GotKey          // Returns True
Drop Drop       // Clean up
RdKey           // Returns the key value
GotKey          // returns False
Drop            // Clean up
WrOut           // Print the character

// Test the Serial connection
GotCom          // Returns False
// Send a character to the the Nest
GotCom          // Returns True
Drop Drop       // Clean Up
RdCom           // Returns the received byte
GotCom          // Returns False
Drop            // Clean up
WrCom           // Send the character back

// Now try it all together
TtyLoop

// Send messages from the sender to the nest
// Check they are displayed correctly on the terminal

// Send messages from the terminal to the connected device
// Check they are displayed correctly on the device

// Any issues review code in the editor, modify and retest before moving on
```

Another Example Application

This next part of the application adds control character support to the earlier dumb terminal example. Control characters are used to mark the end of a line, move the cursor, delete characters and clear the screen. The general ASCII codes presented in the following table are taken from the ANSI terminal protocol.

Table 64. General ASCII Codes

Name	Decimal	Hex	Key	Description
BEL	7	0x07	^G	Rings the bell or sounds a buzzer, only available on devices with audio output.
BS	8	0x08	^H	Moves back one space and erases the character under the cursor.

Name	Decimal	Hex	Key	Description
HT	9	0x09	^I	Moves the cursor to the next TAB stop. Set to 4 by default. Machines with TAB keys emit this code.
LF	10	0x0A	^J	Moves the cursor down one line and scroll the screen up one line if at the last row of the screen.
VT	11	0x0B	^K	Move the cursor to the top left corner of the screen
FF	12	0x0C	^L	Clears the screen and moves the cursor to the top left corner.
CR	13	0x0D	^M	Moves the cursor to column 0 of the current line
ESC	27	0x1B	^[	Escape is used to prefix a sequence of characters that provide extend capabilities. The ANSI escape sequences are documented in the [Serin Terminal] section of Part VI.
DEL	127	0x07F	<none>	Deletes the character under the cursor and shifts the line left one character.

Enhanced Terminal Example

```
// Include the module supporting the serial, keyboard and screen devices
Include "std-io.kf"

Use std-io              // Add it to the extended scope

InputStream comIn       // Now create serial port comms. channels
OutputStream comOut

// Test if character received
( : Flag ) Def GotCom
    com.IsReady
End

// Read a byte from the COM port
( : Byte ) Def RdCom
    comIn.Get
End

// Write a byte to the COM port
( Byte : ) Def WrCom
    comOut.Set
End

// Test if key available
( : Flag ) Def GotKey
    in.IsReady
End

// Read a byte from in
( : Byte ) Def RdIn
    in.Read
End
```

Global Variables

```
Byte posX
Byte posY

( : ) Def InitPosXY
    1 posX.Set
    1 posY.Set
End
```

Control Handlers 1

```
( : ) Def IncrementPosX
    posX.Get line-length =          // If at end of current line
    If
        0 posX.Set                  // go back to start of line
        posY.Get screen-height <    // check if scrolling
        If
            1 posY.Add              // next line
        End
    Else
        1 posX.Add                  // move to next position
    End
End

( : ) DecrementPosX
    posX.Get 0 =        // check if at beginning of a row
    If                  // if so decrement y
        posY 0 >        // check if at top of screen
        If              // if not decrement y
            1 posY.Sub
            line-length posX.Set    // got to end of line
        End
    Else
        1 posX.Sub      // go back one position on same row
    End
End
```

Control Handlers 2

```
( : ) Def BackSpace
    DecrementPosX
    $ ' ' out.WrOut
    DecrementPosX
End

( : ) Def Tab
    // calculate number of spaces required
    posX.Get tabSize.Get % Dup 0 =
    If                  // If zero move 4 more spaces
        Drop tabSize.Get
    End
    0 Swap Range tab    // Create new range
    For tab             // move to Tab stop
        Drop
        IncrementPosX
    End
End

( : ) Def NextLine
    posY.Get  screen-height <
    If
        1 posY.Add
    End
End

( : ) Def ClearScreen
    0 line-length screen-height
    Range screen
    InitPosXY 0L out.Pos
    For screen
        Drop IncrementPosX
        out.Print
    End
End
```

HandleControls

```
( Byte : ) Def HandleControls
    Case
        // Uncomment if audio available and implement RingBell
        // BEL Of RingBell ;
        BS  Of BackSpace ;
        HT  Of NextTab ;
        LF  Of NextLine ;
        VT  Of 1 posX.Set 1 posY.Set ;
        FF  Of ClearScreen ;
        CR  Of 1 posX.Set ;

        // Uncomment the below line and implement EscSequences
        // ESC Of EscSequences ;
        DEL Of BackSpace ;

        // Handle Escape sequence if required
        // ESC Of InitEsc ;
    End

    posX.Get toLong
    posY.Get toLong line-length toLong * +

    // move the cursor to the correct position
    out.SetPos
End
```

Write to Display

```
// Write a byte to out
( Byte : ) Def WrOut
    Dup 32B <   // Is this a control Character
    If
        HandleControls  // Its a ControlChar
    Else
        HandleText      // It's a plain Character
        out.Print
    End
End
```

TTY Loop

```
InitPosXY                                   // Initialise cursor position
out.Pos
// TtyLoop
( : Byte ) Def TtyLoop

    Do
        GotKey
        If                                  // If there is a keypress to process
            RdIn Dup break-key = Not        // Check if the user has pressed the break
key
            If
                WrCom                       // Send it out of the com port
                true                        // Do not exit loop
            Else
                Drop                        // Get rid of break key
                false                       // Terminate the loop
            End
        Else
            true                            // Continue rest of loop
        End
    While
        GotCom                              // Is there character from the Com port?
        If
            RdCom WrOut                     // get the character and write to display
        End
    End
End
```

NOTE

In our original dumb terminal example, the the input and output words were abstracted into new definitions, in this section the reason for that seemingly wasteful abstraction will become apparent. To add control codes to the emulator the following changes are required:

1. Keyboard input; Check that the input words return the control characters as documented in the previous table
2. Display output; Check that the control characters do not have any effect on the screen by default.

Do this using the technique described in the Debug example before continuing.

Part VI: Reference Sections

The Kingfisher system combines modern software development practices with vintage computing concepts to create a robust and flexible development environment. This reference section provides comprehensive documentation of the system's core components, error handling mechanisms, and hardware interfaces.

Each section presents detailed technical specifications, starting with system-level features like error handling and configuration, then progressing through to hardware-specific implementations. The documentation includes practical examples and detailed tables that illustrate key concepts and usage patterns.

Chapter 33. Historical Context

The story of Kingfisher begins with Commodore’s revolutionary approach to personal computing in the late 1970s. Commodore’s acquisition of MOS Technology and their vision of creating "computers for the masses, not the classes" established fundamental patterns that would influence home computing for decades. The PET 2001-8N, with its integrated design and innovative use of the 6502 CPU, set standards for efficient, cost-effective computing that remain relevant today.

Project Aves emerged from a desire to preserve and enhance these proven design principles while embracing modern capabilities. Where Commodore pioneered intelligent peripherals and efficient memory usage, Aves reimagines these concepts using contemporary CMOS components. This approach maintains compatibility with classic architectures while delivering improved efficiency and expanded capabilities.

Origins and Inspiration

The original PET 2001-8N’s integrated design demonstrated how thoughtful hardware architecture could maximise limited resources. Aves builds upon this foundation, replacing original components with modern CMOS equivalents while preserving the elegant simplicity of Commodore’s approach. This careful balance of vintage compatibility and modern enhancement enables Kingfisher to bridge past and present computing paradigms.

Technical Evolution

While Aves began as a hardware-focused initiative to recreate Commodore’s 8-bit systems, it quickly became apparent that software development would play a crucial role. Modern components like SD-Cards and serial memories demanded new interfaces, while preserving compatibility with classic software required careful adaptation of the original architecture. The built-in BASIC interpreter not only lacked support for storage, it also lacked support for the custom silicon, was not easily extended and was very slow. The built in editor was better than most of Commodore’s competitors but it was still very limited. This dual focus on hardware and software development has shaped Kingfisher’s evolution.

Design Philosophy

The design philosophy for Kingfisher emphasises efficiency and purposeful enhancement of proven 8-bit computing principles. The system implements a minimal ROM-based architecture that provides essential system functions while maintaining direct hardware access through memory-mapped I/O. This approach follows established 6502 conventions for memory organisation, particularly in its use of zero-page.

The hardware platform builds upon classic 8-bit architectures through considered enhancement. Modern CMOS components like the 65C02 CPU and 65C22 VIA deliver significant performance improvements, operating at speeds up to 14 times faster than their vintage counterparts. While the 65C02 maintains instruction set compatibility with documented 6502 operations, it includes additional instructions and does not support undocumented or illegal opcodes present in the original processor.

Modern enhancements build upon this foundation through careful implementation of contemporary technologies. Extended I/O capabilities through I2C and SPI interfaces provide expansion options beyond traditional parallel interfaces. Where additional memory addressing is required, the 65816 processor option enables expanded addressing whilst maintaining base architecture functionality.

The development approach emphasises modularity and consistency across the Aves platform family. System components maintain clear interfaces that support hardware expansion whilst preserving core functionality. The architecture ensures immediate system availability upon power-up, with a clear separation between core and extended functionality that enables customisation without compromising reliability.

This balanced approach to system design enables Kingfisher to bridge vintage computing principles with contemporary capabilities. The result maintains the simplicity and directness that characterised early home computers whilst providing a robust foundation for a modern development style. Through careful attention to both historical principles and modern requirements, the system delivers an efficient and flexible platform for 8-bit computing.

Chapter 34. System References

The system references section covers fundamental aspects of Kingfisher's operation, beginning with error handling mechanisms, followed by system configuration options, and concluding with system constants. These components form the foundation for reliable application development.

Error Message Reference

This section provides a comprehensive reference for all system error messages in Kingfisher. Errors are categorised by their functional area and severity, allowing for systematic error handling in applications. Each error is uniquely identified by a hexadecimal code and belongs to a specific category range.

The first table shows the error categories and their hexadecimal ranges. The second table provides detailed information for each error, including its message, the condition that triggers it, and recommended recovery actions.

Table 65. Error Categories

Symbolic Name	Range	Category Name
EC-ARITH	0x00-0x1F	Arithmetic Operations
EC-STACK	0x20-0x3F	Stack Operations
EC-DICT	0x40-0x5F	Dictionary Operations
EC-SYS	0x60-0x7F	System Resources
EC-IO	0x80-0x9F	I/O Operations
EC-MEM	0xA0-0xBF	Memory Management
EC-COMP	0xC0-0xDF	Compilation
EC-RSVD	0xE0-0xFF	Reserved

Table 66. Error Message Reference

Code	Category	Error Message	Recovery Action
0x00	EC-OK	Success	
0x01	EC-ARITH	Divide by zero	Implement OnError block to handle division by zero: return a default value, log the error, or propagate to caller
0x60	EC-SYS	End of iteration	Terminate the loop and stop iteration
0x42	EC-DICT	Index out of bounds	Add bounds checking before access: verify index < collection.size. Consider using safe access methods
0xC1	EC-COMP	Local name > 8 chars	Shorten the name
0x21	EC-STACK	Stack overflow	Increase stack depth, and check stack signatures

Code	Category	Error Message	Recovery Action
0x20	EC-STACK	Stack underflow	Add stack depth checking
0x43	EC-DICT	String not found	Implement recovery block to either; create missing string, use default value, or terminate with error message
0xC2	EC-COMP	Type mismatch	Ensure type compatibility
0x41	EC-DICT	Unknown Word	Check word exists before execution, or implement word lookup fallback mechanism in current scope
0xC0	EC-COMP	Unsupported type	Change type before calling the definition
0xC3	EC-COMP	Value out of range	Select a different type
0x40	EC-DICT	Word exists in this scope	Choose unique name
0xC4	EC-COMP	Word is compile only	Use word inside a definition
0x80	EC-IO	Device not responding to requests	Check drive power and connections
0x81	EC-IO	Filesystem unavailable	Mount a valid filesystem with the mount command
0x82	EC-IO	Disk Error: <drive status message>	Consult the manufacturers drive documentation
0x83	EC-IO	Disk Write protected	Remove write protection or try another disk
0x84	EC-IO	File not Found	Check the file name and try again
0x85	EC-IO	File exists	Change the file name or file open mode
0x86	EC-IO	File Error <file-error-message>	Consult the manufacturers drive documentation
0x40	EC-DICT	Dictionary Full	Remove unused vocabularies from dictionary
0x41	EC-DICT	Word does not exist	Check spelling of name or make sure word exists
0x42	EC-DICT	Name Exists	Change spelling of new word
0xA0	EC-MEM	Out Of Memory	Review memory usage, reduce string and array sises. Remove unused vocabularies

Chapter 35. Technical Background

This section provides comprehensive documentation for the Kingfisher development environment. This guide details the core components, methods and patterns that enable effective software development within the Aves platform ecosystem. Kingfisher implements its own stack-based programming language, providing a fresh approach to 8-bit system development.

The technical architecture reflects this heritage whilst incorporating considered enhancements. At its core, Kingfisher maintains compatibility with classic hardware paradigms, particularly the Commodore approach to device management and memory organisation. This design philosophy enables the system to support both vintage hardware and retro 8-bit Aves platforms through a unified interface. The range will also be expanded to include a range of 16 bit models.

The system architecture implements established conventions from the 8-bit era, such as device numbering and memory mapping, whilst updating the architecture with expanded I/O capabilities and enhanced memory management. This balance of traditional and enhanced features creates a robust foundation for classic stack-based software development.

Through its modular design, Kingfisher accommodates various hardware configurations across the Aves family of systems. The architecture provides consistent interfaces for peripheral access, memory management, and system resources, allowing developers to create portable applications that function across multiple platforms within the Aves ecosystem.

System Architecture

The Kingfisher system architecture builds upon established 8-bit computing principles, emphasising simplicity and efficient resource utilisation. Early home computers established fundamental patterns through ROM-based firmware and embedded language systems, an approach that proved both reliable and effective. Kingfisher extends these principles whilst maintaining compatibility with vintage software and hardware designs.

Operating System Design

The Kingfisher operating system maintains a similar pattern to the early home computers. Core functionality resides in ROM, providing reliable operation and immediate availability upon system startup. The ROM contains the kernel, compiler, interpreter, editor and command line interface. Direct hardware access occurs through memory-mapped I/O, following established patterns for device interaction. Additional features load from external storage when required, enabling system customisation without compromising core functionality.

The modular design supports hardware expansion through multiple serial protocols, enabling the use of a wide variety of different hardware configurations. This approach maintains the simplicity and reliability inherent in ROM-based systems.

Hardware Implementation

The Aves hardware platform implements core system functionality through carefully selected enhancements to vintage designs. Custom video and audio subsystems maintain software

compatibility whilst providing expanded capabilities. CMOS variants of classic processors deliver improved efficiency whilst preserving instruction set compatibility. The following [Processor Options] table show the processors provided by Aves and supported by Kingfisher

Table 67. Processor Options

Processor	Implementation
6502	Standard NMOS instruction set
65C02	Enhanced CMOS instruction set
65816	Extended 16-bit instruction set

Input/Output System

Device management implements the Commodore convention for peripheral addressing, ensuring straightforward integration with existing software. The system supports both traditional peripherals and enhanced expansion options through a consistent device numbering scheme.

Table 68. Device Assignments

Device	Function
0	Keyboard input
1	Cassette interface
2	RS232 and I2C UART functions
3	Display output
4-5	Printer operations
6	I2C expansion interface
7	SPI expansion interface
8-15	Disk operations

Memory

The Kingfisher system implements a flexible memory architecture that builds upon established 8-bit computing principles. This organisation provides efficient usage of ROM and RAM, and provides sufficient capacity for both ROM and RAM based applications.

Memory management divides the 64K address space into distinct functional regions. System RAM occupies the lower 32K, providing workspace for applications and system variables. The 6502 stack is fixed at page 1, and is used for subroutine and interrupt return addresses and saving registers. Zero page is used for the parameter stack, indirect pointers, and system variables.

The upper memory regions contain system firmware, I/O device registers and boot code. This arrangement maximises the amount of available memory without restoring to complex RAM banking or paging operations. The firmware region houses the Kingfisher implementation alongside essential system routines, whilst dedicated I/O space provides consistent peripheral access.

The Aves 65C02 models do not use any memory banking, but other models with more memory use the 65816's segmented 24 bit address capability to address more than 64K RAM, however the memory map in segment 0 remains consistent.

Table 69. Memory Capacity

Memory Space	Sises	Usage
System ROM	32K EPROM 128K FLASH	Firmware and system routines
Main RAM	32K 128K 512K	Application workspace
Extended Storage	128K 256K	I2C EEPROM (optional)
Expansion	SD-CARD etc.	I2C/SPI interfaces

NOTE Extended storage availability depends upon hardware configuration and platform implementation.

Table 70. 64K System Memory Map

From	To	Usage
0x0000	0x7FFF	RAM - System RAM
0x8000	0xF7FF	EPROM - Firmware, Kingfisher, Talon, Kernel
0xF800	0xFBFF	I/O
0xFC00	0xFFFF	EPROM - Memory test and boot

NOTE Zero page occupies 0x0000-0x00FF, with system stack at 0x0100-0x01FF.

Memory Map Example

```
0x8000 Constant ROM-BASE    // Start of system ROM
0x0000 Constant RAM-BASE    // Start of system RAM
0x7FFF Constant RAM-TOP     // Top of standard RAM
```

Vintage Operating Systems

The first generation of home computers did not have operating systems like modern day operating systems. Instead manufacturers provided ROM based firmware and an embedded language, usually BASIC. The likes of Commodore, Apple, Atari and Acorn adopted this approach to great effect and also published software on ROM based expansion cartridges. In fact many of these systems did not have disk drives and relied on tape drives to save programs and data.

Commodore pioneered the idea of off loading the work of running the Disk Operating System (DOS) onto a separate device but did not provide any form of filesystem abstractions in most of the BASIC

Interpreters due to memory limitations.

Development References

The Development Reference section provides comprehensive documentation for the Kingfisher development environment. This guide details the core components, methods and patterns that enable effective software development within the Aves platform ecosystem, with particular focus on Kingfisher's unique stack-based programming environment.

Kingfisher Lexicon

A categorised index of all of Kingfishers bespoke words can be found in the [Kingfisher Words by Category] section below.

Table 71. Kingfisher Words by Category

Reference	Category
[Stack Words]	Stack manipulation words
[Arithmetic Words]	
[Bitwise Words]	
[General Words]	
[Type Conversion Words]	
[Variable Constructors]	Variables
[Variable Methods]	
[Array Constructors]	Collections
[Array Methods]	
[Typed Array Methods]	
[StrArray Constructors]	
[StrArray Methods]	
[Collection Iteration Words]	
[Bootstrap Words]	Bootstrap, Scope and Vocabulary
[Chain methods]	
[Vocabulary Constructor]	
[Module Words]	
[Alias Word]	

Reference	Category
[Type Definition Words]	Type definition
[Scalar Vocabulary Words]	
[ShortSeq Vocabulary Words]	
[Sequence Vocabulary Words]	
[Type Field Words]	
[Dataset Methods]	
[Boolean Words]	Boolean and conditional operators
[Conditional Words]	
[If Branch Operations]	Branches
[Case Branch Operations]	
[While Loop Words]	Loops
[For Loop Words]	
[Error Handling Words]	System
[Error Methods]	
[Fundamental System Words]	
[Dictionary Words]	
[Heap Management Words]	
[Label Word]	Assembly
[Data Definition Directives]	
[Section control]	
[Assembly Language Words]	
[System Data Constructors]	
[System Data Methods]	
[Stream Constructors]	Stream IO
[Stream IO Methods]	
[File System Methods]	File IO
[Volume Methods]	
[File Methods]	

Perch CLI Reference

The Perch Command Line Interface provides interactive access to the Kingfisher development environment through a REPL (Read-Eval-Print Loop). It combines traditional command line functionality with enhanced features for command editing and history navigation, whilst maintaining compatibility with vintage keyboard layouts.

The interface preserves commands in a 256-byte buffer, allowing developers to recall and modify previous entries. Command editing capabilities include cursor movement, character insertion and deletion, and line manipulation functions.

Table 72. Command Line Features

Feature	Description
History Buffer	256-byte buffer for command storage
Line Editing	Full cursor movement and text manipulation
Command Recall	Forward and backward history navigation
Character Support	Mapped special characters for vintage keyboards

Table 73. Control Key Mappings

Command	Ctl Code	CBM Key	PC Key	Function
Backspace	^H	INST/DEL	Backspace	Delete the character to the left of the cursor and move left one space
Break	^C	RUN/STOP	—	Stop the running program (can be overridden)
CrsrDown	^S	CRSR UP/DN	Crsr Up	Move cursor down one space
CrsrEnd	^E	—	End	Move cursor to end of line
CrsrHome	^Q	CLR/HOME	Home	Move cursor to beginning of line
CrsrLeft	^A	CRSR Lt/Rt	Crsr Left	Move cursor left one space
CrsrRight	^D	CRSR Lt/Rt	Crsr Right	Move cursor right one space
CrsrUp	^W	CRSR UP/DN	Crsr Down	Move cursor up one space
DelChar	DEL	—	Del	Deletes the character under the cursor
DelLine	^L	CLR/HOME	—	Clear the entire line
InsMode	INST	INST/DEL	Ins	Toggles insert mode (default is on)

Usage Example

```
> 10 Constant Test ok    // Define constant then prompt ok
> Test Print 10 ok       // Print the test value
> // Recall previous command with :^W
Test Print
```

Talon IDE Reference

The Talon IDE provides a full-screen text editor optimised for the Kingfisher development environment. It combines efficient screen usage with flexible editing capabilities while respecting the constraints of vintage hardware. The editor supports both vertical and horizontal scrolling, with configurable margins to maximise usable screen space on displays ranging from 28 to 100 columns wide.

The interface balances functionality with simplicity through a two-line menu system. The top menu line presents command categories, while the second line provides contextual descriptions of available operations. All editor functions are accessible through both menu selection and direct control key commands, accommodating different user preferences and keyboard layouts.

Text manipulation features include both character-level horizontal selection and line-level vertical selection, enabling precise editing control. The display automatically manages available space between the edit area, debug panel, and menu overlay to maintain optimal visibility of the working text. Vertical scrolling operates line by line, while horizontal scrolling moves by screen width with configurable margins up to 100 character line lengths.

The following tables provide comprehensive references for editor commands and display specifications across the range of supported hardware configurations.

While Commodore machines traditionally use PETSCII encoding, Kingfisher implements standard ASCII character encoding. This provides better compatibility across the Aves platform family whilst maintaining consistent text representation. On the Commodore 64 and Aves machines, CodePage 437 is also available, offering extended character support. The PETSCII graphics character set is not currently supported.

It should be noted that on Commodore hardware, the uppercase and lowercase character sets are transposed compared to standard ASCII - uppercase characters occupy the codes normally used for lowercase and vice versa. Kingfisher handles this transparently, ensuring consistent text display across all supported platforms.

Table 74. Editor Commands

Command	Menu Item	Ctrl Key	Description
Backspace	—	^H	Deletes the character to the left of the cursor, and moves the cursor left one space
CrsrDown	—	^S	Move cursor down one space. Also cursor down key
CrsrEnd	—	^E	On first press move cursor to end of line, on second press go to bottom right of display
CrsrHome	—	^Q	on first press move cursor to beginning of line, on second press go to top left of display. Or go directly to the top left by pressing the home key
CrsrLeft	—	^A	Move cursor left one space. Also cursor left key
CrsrRight	—	^D	Move cursor right one space. Also cursor right key
CrsrUp	—	^W	Move cursor up one space. Also cursor up key
DelLine	—	^L	Clear the entire line. Press again clears the screen. Or clear the entire screen with the clr key
DelChar	—	INST/DEL	Deletes the character under the cursor

Command	Menu Item	Ctrl Key	Description
EditBegin	edit-begin Menu-E-B	^B	Marks the beginning of an area of text
EditCopy	edit-copy Menu-E-C	^C	Copy the marked text into a temporary buffer
EditE£xtract	edit-extract+ Menu-E-X	^X	Extract the marked text into a temporary buffer
EditEnd	edit-end Menu-E-N	^N	Marks the end of a text area
EditPaste	edit-paste Menu-E-V	^V	Paste the text from the temporary buffer into the editor
FileNew	file-new Menu-F-N	—	Create an empty file and clear all edit buffers
FileInsert	file-insert Menu-F-I	—	Insert an existing file into memory starting at the line before the current line
FileOpen	file-open Menu-F-O	—	Load an existing file int0 memory replacing existing contents
FileSave	file-save Menu-F-S	—	Backup the existing file to a new name and save the current file
FileClose	file-close Menu-F-C	—	Close the file and clear memory
FileExit	file-exit Menu-F-X	—	Checks if all changes have been saved and exits the editor
SearchFind	search-find Menu-S-F	^F	Search file from current position until a match is found
SearchGoto	search-goto Menu-S-G	^G	Goto a specific line number
SearchNext	search-next Menu-S-N	^N	Moves to the next matching pattern
SearchOptCase	search-opt-case Menu-S-O-C	^O	Toggle case sensitivity option
SearchOptGlobal	search-opt-global Menu-S-O-G	^O	Toggle global replace option
SearchOptWord	search-opt-word Menu-S-O-W	^O	Toggle match complete word only option
SearchReplace	search-replace Menu-S-R	^R	Replace matched patterns with replacement text

Command	Menu Item	Ctrl Key	Description
ViewDebug	view-debug Menu-V-D	—	Switch to Debug View
ViewEdit	view-edit Menu-V-E	—	Switch to Editor View
ViewSplit	view-split Menu-V-S	—	Switch to Split View, which shows a view of the editor in the upper part of the screen and the REPL in the lower part
ViewSwitch	view-toggle Menu-V-T	^T	Toggle which view; Debug or Edit
OptionsConfig	option-config Menu-O-C	—	Configure options

Table 75. Display Specifications

Model	Width	Height	Colours
Commodore PET	40	25	Mono
Commodore CBM-80XX	80	25	Mono
Commodore VIC-20	28	30	8 Colours
Commodore C64	40	25	16 Colours
Aves Sparrow	50	30	16 Colours
Aves Dunnock	100	30	Mono
Aves Robin	28	28	8 Colours
Aves Blackbird	40	25	16 Colours
Aves Starling	40	25	16 Colours

Table 76. Keyboard Mapping

Model	Menu	TAB	Control	Delete	Backspace	Insert
PET	←	—	Off/RVS	—	INST/DEL	SHIFT + INST/DEL
CBM 80XX		Tab		Run/Stop		
VIC-20	←	Ctrl	C=	F1		F3
C-C64						
Aves Sparrow	Esc	Tab	Control	Del	Backspace	Ins
Aves Dunnock						
Aves Robin						
Aves Blackbird						
Aves Starling						

Table 77. Configuration options

Option	Value	Description
Margin size	0-20	Horizontal scrolling margin (default=15, disable=0)
Editor window	(screen-height-4)-5	The line where the editor/debug windows split

Chapter 36. Fundamental System Structures

Kingfisher's architecture builds upon several core structures that work together to create a robust and extensible system. These structures provide the foundation for the language's key features: compile-time type checking, efficient word lookup, and runtime safety. Understanding these fundamental structures is essential for both using and extending the system effectively.

Dictionary Entries

The dictionary is the heart of Kingfisher's extensible architecture, storing all definitions, variables, and constants. Each dictionary entry forms part of a linked list, allowing for efficient lookup while maintaining the system's extensibility. To ensure type safety during compilation, each entry includes a type signature that encodes the stack signature using type modifiers, enabling robust compile-time type checking without runtime overhead.

NOTE	Dictionary entries can be aligned on machine word boundaries to ensure optimal access performance, for 16 bit CPU architectures.

Dictionary Entry Structure

Field	Description
Type Signature	Variable length byte-counted array of type modifiers (0-6 bytes). Each modifier encodes input/output status and type information for stack signature checking.
Name	Variable length byte-counted string with maximum length of 20 bytes. Case-sensitive identifier used to reference the entry. The upper bits of the length byte are reserved: **Bit 7 Immediate:** When set, the word executes immediately during compilation rather than being compiled into the definition. e.g. The word `Def` is marked Immediate so it can create new definitions while compiling. **Bit 6 CompileOnly:** When set, the word can only be used during compilation. Attempting to execute these words directly will raise an error. e.g. Control flow words like `If`, `Else`, and `End` are marked CompileOnly since they only make sense during compilation.
Link	Pointer to the previous dictionary entry, creating a linked list structure that enables lookup.
Code	Executable machine code. For STC, contains subroutine calls that can be optimised by replacing the final call with a jump. For DTC, contains the threaded code sequence.
Data	Pointer to associated data schema or size of storage required. The field is empty if no data required.

Type System

The type system provides compile-time type checking through distinct but complementary mechanisms. Type signatures provide human-readable representation of stack effects, while type modifiers encode this information for the compiler. The system defines fundamental types as building blocks and supports signature literals for type-aware dictionary operations.

Base Types

The system defines a set of fundamental types used throughout the language. These form the building blocks for all type operations.

Table 78. Base Types

Value	Type	Description
00	Var	Default type for numeric values and primitive operation addresses
01	Byte	Unsigned 8-bit value for bit operations and characters
02	Word	16-bit unsigned integer value
03	Sword	16-bit signed integer value
04	Long	32-bit integer value
05	Flag	Boolean value
06	String	Character string reference
07	-	Reserved for future use
08-31	User defined types	Application-specific type definitions

Type Signatures

Type signatures provide a human-readable representation of stack effects for words. The signature notation follows the natural stack order, where parameters are read from right to left, matching the order of stack operations. Input parameters precede output parameters, separated by a colon.

Type Signature Examples

```
// A word consuming a Var then a Byte, yielding a Flag
( Byte Var : Flag )

// A word yielding a String without input parameters
( : String )

// A word consuming two Words, yielding a Word
( Word Word : Word )
```

Type Modifiers

Type modifiers encode type information in a binary format for compiler use. Each modifier byte contains flags and type identifiers that fully specify parameter characteristics. The compiler utilises these modifiers during type checking and word compilation.

Table 79. Type Modifier Format

Bit Position	Name	Description
7	I/O	Input (0) or Output (1) parameter
6	Array	Single value (0) or Array type (1)
5	Slice	Full value (0) or Reference/view (1)
4-0	Base Type	Type identifier (0-31)

Type Signatures and Modifiers Examples

```
// Type Signature with:
// Inputs: Byte Var     -> Consumed
// Outputs: Flag        -> Produced
( Byte Var : Flag )     // => 0x03 0x01 0x00 0x85

// Type Signature with:
// Inputs: None
// Outputs: String      -> Produced
( : String )            // => 0x01 0x85

// Type Signature with:
// Inputs: Word Word    -> Consumed
// Outputs: Word        -> Produced
( Word Word : Word )   // => 0x03 0x02 0x02 0x82
```

NOTE | See [dictionary-concepts] for details about vocabularies and namespaces.

The Dictionary

The Dictionary is the core component of Kingfisher, managing all definitions, types, and variables. It provides fast name lookup and manages memory allocation. This core subsystem forms the foundation for Kingfisher's extensible nature.

Dictionary Concepts

The Dictionary implements an efficient symbol management system focused on compile-time type checking and name resolution. While it supports basic runtime operations, its primary role is enabling robust compile-time verification of types and symbol visibility. The system emphasises compile-time checks over runtime flexibility to ensure reliable and efficient execution.

At its core, the Dictionary maintains entries for every symbol in a Kingfisher program. Each entry

can represent a variable holding runtime data, a type definition that defines data structures, or a function or procedure. Dictionary entries also store constants with fixed values, and manage modules and namespaces that provide code organisation. These different kinds of entries share a common structure while serving distinct roles in the system.

Multiple mechanisms are employed to manage these symbols throughout their lifecycle. Symbol resolution forms the cornerstone of the Dictionary's operation, providing the means to locate and access program elements across different scopes. Working in concert with the memory management subsystem, the Dictionary coordinates the allocation and deallocation of resources, ensuring efficient use of system memory while maintaining program safety.

Symbol Resolution

Symbol resolution in Kingfisher operates through a multi-stage process. When resolving a symbol, the Dictionary first traverses the scope hierarchy, beginning with the local scope and progressively moving outward toward the global scope until the symbol is found. This hierarchical search respects symbol visibility rules that determine which scopes can access particular symbols, enforcing encapsulation and overlaying inner scope names over outer scopes. During resolution, the Dictionary also validates type information, ensuring type compatibility and gathering necessary metadata about the symbol's structure and behaviour. The Dictionary entry's data field is set to point to this allocated storage, while the code field contains executable machine code - either subroutine calls that can be optimised by replacing the final call with a jump (for STC), or the threaded code sequence (for DTC).

Dictionary Reference

The dictionary forms the foundation of Kingfisher's extensible architecture, implementing word storage, type checking, and runtime safety.

Dictionary Entry Structure

Each dictionary entry contains five fields:

Table 80. Dictionary Entry Fields

Field	Description
Type Signature	Variable length counted array of type modifiers (0-6 bytes) encoding stack effects
Name	Variable length counted string (max 20 bytes) with control bits: * Bit 7 (Immediate): Executes during compilation * Bit 6 (CompileOnly): Only valid during compilation
Link	Pointer to previous dictionary entry
Code	Executable machine code (STC or DTC format)
Data	Pointer to static data storage (0 if none)

Type System Integration

Dictionary entries use type signatures for compile-time checking:

Type Signature Format

```
( input-types : output-types )

Examples:
( Byte Var : Flag )     // Consumes Byte,Var; produces Flag
( : String )            // Produces String
( Word Word : Word )    // Consumes two Words; produces Word
```

Table 81. Base Types

Value	Type	Description
00	Var	Default numeric type and primitive addresses
01	Byte	Unsigned 8-bit value
02	Word	16-bit unsigned integer
03	Sword	16-bit signed integer
04	Long	32-bit integer
05	Flag	Boolean value
06	String	Character string reference
07	-	Reserved
08-31	User	Application-specific types

Type Modifiers

Each modifier byte encodes parameter characteristics:

Table 82. Type Modifier Format

Bit	Name	Description
7	I/O	Input (0) or Output (1) parameter
6	Array	Single value (0) or Array type (1)
5	Slice	Full value (0) or Reference/view (1)
4-0	Base Type	Type identifier (0-31)

Dictionary Management

The Dictionary provides a comprehensive set of operations for managing symbols and their associated data throughout the system lifecycle. These operations handle everything from symbol creation and lookup to type checking and memory management. The Dictionary's management functions are designed to maintain system integrity while providing efficient access to program

elements.

Dictionary Structure

The Dictionary follows the singleton pattern, ensuring exactly one instance exists throughout the system. This single instance is created during system initialisation and maintains the global symbol table. The dictionary is constructed at compile time with access to the Dictionary provided through static instance methods. This design ensures consistent symbol management and prevents multiple competing Dictionary instances from being created.

Memory Management

Kingfisher's memory management system provides two primary abstractions for working with memory through Blocks and Slices. These abstractions ensure safe and efficient memory handling whilst preventing common issues such as buffer overflows and memory leaks.

Memory Blocks

Memory blocks serve as the foundation of Kingfisher's memory management system through fixed-size memory allotments. Each block maintains internal size information and implements bounds-checked access to its contents. The system manages the freeing of blocks automatically and ensures proper memory alignment for efficient access.

NOTE Memory blocks are allotted and freed when scopes are opened and closed, simplifying the memory management strategy.

Memory Slices

A slice creates a view into an existing memory block without copying the underlying data. When created, a slice establishes a reference to its source block along with position information comprising an offset and length. This approach allows the slice to define a specific region within the block.

The slice mechanism ensures memory safety through comprehensive bounds checking and reference counting of the memory slices. This design enables efficient memory access through zero-copy operations whilst maintaining complete memory safety.

NOTE Memory Slices can only reference objects in the current scope, avoideing the need for complex memory management strategies

Compile-time Features

Kingfisher implements comprehensive compile-time checking and validation to ensure program correctness before execution. The system performs thorough type analysis, manages definitions, and resolves symbols across different scopes whilst maintaining strict safety guarantees.

Type Checking

The compiler performs exhaustive type checking during compilation, by tracking stack signatures on a type stack and checking against the embedded type modifiers, ensuring type safety. When the compiler encounters a word usage, it compares the current stack state with the word's type signature to verify compatibility.

Type checking encompasses validation of array and slice operations to ensure type safety. The type system verifies that operations maintain the correct types when working with arrays and slices at compile-time, while actual bounds checking occurs at runtime.

NOTE	Runtime bounds checking provides an essential safety mechanism for array and slice operations. The type system ensures type safety while runtime checks prevent buffer overflows and invalid memory access.

Definition Management

The Kingfisher compiler maintains a dictionary of word definitions during compilation. Each word in the dictionary includes its name, compilation behaviour, and visibility scope. This dictionary serves two primary purposes: it enables the compiler to locate word definitions when they are referenced, and it enforces the rules about how and when words may be used.

Some words require special handling during compilation. Immediate words execute when encountered during compilation, making them useful for control structures and compile-time operations. Compile-only words can only appear within definitions and cannot be executed in the interpreter. The compiler enforces these restrictions by checking each word's attributes when it is used.

NOTE	The dictionary system allows the compiler to detect undefined or misused words during compilation, preventing runtime errors that would occur if missing or incorrectly used words were discovered during execution.

Symbol Resolution

Symbol resolution occurs during compilation through a multi-stage process that determines the correct interpretation of each identifier. The compiler searches through nested scopes to locate definitions, starting from the innermost scope and proceeding outward to the global scope. This process ensures that local definitions take precedence over global ones, whilst maintaining access to outer scope definitions when needed.

NOTE	The resolution process maintains consistent naming within each scope whilst preserving the expected precedence rules of nested definitions.

Module System

The module system organises code into separate compilation units called modules. Each module creates its own vocabulary of definitions that are public by default. When a module is included, its definitions are compiled but its vocabulary remains excluded from the active search chain.

The Use command adds a module's vocabulary to the extended search chain. This mechanism allows definitions in the extended search chain to hide matching names in outer scopes.

NOTE	The vocabulary system provides namespace isolation between modules while allowing controlled access through Use commands.

Index

@

(KF), 1
(KPL), 1
.D64, 5

A

Addressing Modes, 89
alias, 56
Aliases, 56
Assembly Directives, 90

B

BIDMAS, 15
BODMAS, 15
bootstrap, 53
Bounds Checking, 3

C

CompileOnly, 149

D

definitions, 47

I

IDE, 122
Immediate, 149
Infix, 15
Instruction Set, 87
Integrated Development Environment, 122

K

Kingfisher Programming Language, 1

L

Last-In-First-Out (LIFO) stack, 3
linked methods, 29
load, 53
Locate, 95

M

Macro Processing, 92
memory allocation, 3
module, 56
Modules, 56

P

PEMDAS, 15
Postfix, 15
program, 53

R

Register, 95
REPL
    CLI, 13
    using the repl, 13
reverse Polish notation, 15
ROM Cartridge, 5

S

Save, 54
SDLC, 121
strong typing, 3

T

The Software Development Lifecycle, 121

V

VICE Emulator, 5

W

word, 13, 47-48, 125