Extensions To The Flight Programming Language

Charles Eric LaForest

December 2005
University of Waterloo
Independent Studies

Abstract

The Flight programming language is based on a small kernel of primitives sufficient to linearly allocate memory, compile stack-based machine code, associate a name to a memory address, look up a memory address from its name, and execute the code there. Although self-contained, this kernel lacks useful features such as the freeing of allocated memory, interactive use of the underlying machine, inline compilation of code and strings, string and number output, flow control constructs, elementary multiplication and division, function composition, lexical closures, and higher-order functions. These features are added to Flight without additions the kernel and without the use of software tools other than Flight itself. The code for these features adds up to about 300 lines of source and compiles into about 2000 words of 32-bit memory.

Contents

1 Introduction

The Flight programming language kernel [LaF05b] is an attempt at a minimal self-extensible language suited to the Gullwing second-generation stack-based computer architecture [LaF05a] which allows more sophisticated programming than a symbolic assembler and more expressive power than a traditional compiler while being simpler to implement.

While the kernel of Flight meets these goals it lacks many features. Memory, once allocated, cannot be freed. There is no way to interactively read and write from memory and evaluate expressions. There are no inline versions of functions. There are no means to print strings and decimal numbers. There are no flow control constructs except for the direct use of conditional jumps. There are no arithmetical operations other than addition. There is no function composition with arbitrary numbers of return values and parameters. There are no lexical closures or higher-order functions to make programming more succinct and powerful.

While the Flight kernel itself could be modified to support these features, it would entail a lot of tedious bug-prone assembly programming and would burden the kernel with code that is not essential. Flight is its own interpreter, assembler and compiler and thus it is possible to implement the aforementioned features not as monolithic assembly primitives, using external tools, but as a hierarchy of functions compiled to efficient machine code and described in the Flight language itself.

1.1 Gullwing Programmer Model Summary

The Flight programming language has many functions which directly relate to the instructions of the Gullwing processor. Here is a summary programmer's view of the underlying hardware.

1.1.1 Data Stack

The data stack is a 32-bit wide, 16-deep Last-In-First-Out buffer where most of the computations are performed. The instructions which operate upon it are:

LIT	pushes a literal from the next memory location
XOR	replaces the two topmost elements with their bit-wise XOR
AND	replaces the two topmost elements with their bit-wise AND
NOT	replaces the topmost element with its binary complement
2*	shifts the topmost element one bit to the right, shifting a zero into the least significant bit
2/	shift the topmost element one bit to the left, duplicating the sign bit
+	replaces the two topmost elements with their sum
+*	adds the second element to the first if the first element is an odd number (multiply step)
DUP	pushes a copy of the topmost element
DROP	pops the topmost element
OVER	pushes a copy of the second element

1.1.2 Return Stack

The return stack is a 32-bit wide, 16-deep Last-In-First-Out buffer where function call return addresses are kept. It is also used for sequential memory access and temporary storage. The instructions which operate upon it are:

CALL	branches to an address stored in the next memory location and pushes the return address
RET	pops the return address and branches to it
JMP	jumps unconditionally to an address stored in the next memory location
JMP0	pops the topmost element of the data stack, jumps to an address stored in the next memory location if it is equal to zero
JMP+	pops the topmost element of the data stack, jumps to an address stored in the next memory location if it is positive
R@+	pushes onto the data stack the content of an address on the top of the return stack, post-increments the address by one
R!+	pops the topmost element on the data stack and stores it at the address on the top of the return stack, post-increments the address by one
>R	pops the topmost element on the data stack and pushes it onto the return stack
R>	pops the topmost element on the return stack and pushes it onto the data stack

1.1.3 Address Register

The 32-bit wide address register is the main means of memory access. The instructions which operate upon it are:

>A	pops the topmost element of the data stack and stores it into the address register
A>	pushes a copy of the address register onto the data stack
A@	fetches from the address in the address register and pushes the data onto the data stack
A!	pops the topmost element of the data stack and stores it at the address in the address register
A@+	fetches from the address in the address register and pushes the data onto the data stack, post-increments the address by one
A!+	pops the topmost element of the data stack and stores it at the address in the address register, post-increments the address by one

1.1.4 Other

There are six instructions not tied to the Stacks or the address register:

NOP	Does nothing but consume a processor cycle.
UNDEF[0-3]	four opcodes which are still unused and are currently defined as NOPs
PC@	Fetches the next group of instructions into the Instruction Shift Register. There are six instruction slots or one literal per memory word. This instruction is invisible to the programmer and compiled automatically by the `ALIGN` kernel function.

1.2 Flight Internals Summary

A summary description of the internal structure of Flight will make the extensions presented easier to understand.

1.2.1 Counted Strings

The elementary unit of memory allocation is the counted string. The head of a string is a single memory word which holds the number of memory words used by the contents of the string, which follow the head and are arbitrary. The tail of the string is an additional memory word after the contents which holds its own address.

The head and the tail make it easy to copy, process, and compare strings. Copies and calculations are accelerated by knowing the length of the string in advance. Comparisons fail immediately if two strings are of different length and necessarily end at the tail, which is unique to each string.

1.2.2 Code Dictionary

The code dictionary is where machine code is compiled. It begins at the first address in memory and expands towards the last. Instructions are compiled one after the other without delimiters. The current and next compilation addresses are held in the HERE and HERE_NEXT kernel variables.

1.2.3 Name Dictionary

The name dictionary is where the names and addresses of function are kept. It begins near the last memory location, after some input/output ports, and expands towards the first. Name entries are modified counted strings, appended one after the other. The content of the string is a name associated with a memory location. The tail of the string contains the address of the memory location associated with the name. This address is usually the beginning of the code for a function. The name dictionary is searched in reverse order, so the latest matching name will be the one found. The address of the head of the latest entry in the name dictionary is kept in the THERE kernel variable.

1.2.4 Input Buffer

The input buffer begins right before the latest name dictionary entry and encompasses all of the unused memory up to the tail of the code dictionary. The address of the current head of the buffer is stored in the INPUT kernel variable. Counted strings are allocated and consumed on the buffer in a stack-like manner. If there is only one string in the buffer, it can be converted to a name entry in place by storing the code address in the tail and altering THERE to point to the head of the string and INPUT to point to the first memory location before that string.

2 Extensions To The Flight Kernel

The extensions are presented in logical order and assume a bare Flight kernel. The most important detail to keep in mind is that the kernel does not parse code like a compiler, but looks up and executes words as they are presented. Compilation is achieved by the execution of words which compile code into memory.

2.1 Compilation

The first thing implemented is an easier way to define and call functions, which is used throughout the later kernel extensions.

2.1.1 Defining Function Names

```
SCAN : DEFN SCAN SCAN LOOK CMPCALL SCAN DEFN LOOK CMPCALL CMPRET
```

This code is executed as follows:

1. `SCAN` reads the string ":" into the input buffer.
2. `DEFN` creates a name entry with that string, pointing to the current location in memory where code is to be compiled.
3. `SCAN` reads in "SCAN".
4. `LOOK` looks up the code address of "SCAN" and leaves it on the data stack.
5. `CMPCALL` compiles a function call to the address on the data stack.
6. `SCAN` reads in "DEFN".
7. `LOOK` looks up the code address of "DEFN" and leaves it on the data stack.
8. `CMPCALL` compiles a function call to the address on the data stack.
9. `CMPRET` compiles a function returns.

The resulting machine code resembles the following assembly code:

```
call SCAN
call DEFN
ret
```

The net result is a new word, `:`, which will read in the following string and associate a memory address with it.

2.1.2 Looking Up Functions By Name

```
: l SCAN SCAN LOOK CMPCALL SCAN LOOK LOOK CMPCALL CMPRET
```

This Flight code compile this machine code:

```
call SCAN
call LOOK
ret
```

which is a function named `l` which reads in the following string and looks up the memory address associated with it.

2.1.3 Compiling Function Calls

Flight code:

```
: $c l LOOK CMPCALL l CMPCALL CMPCALL CMPRET
```

Resulting machine code:

```
call LOOK
call CMPCALL
ret
```

which is a function named `$c` which compiles a call to a function whose name is in the input buffer.

It is now possible to compile a call to a preexisting function foo by inputting the following Flight code:

```
SCAN foo $c
```

Some Flight code, `SCAN` in this instance, places the name of a function in the input buffer and cause a call to be compiled to it. This is already much shorter than what the bare Flight kernel allows, but it can be improved further:

```
: c SCAN SCAN $c SCAN $c $c CMPRET
```

This integrates the use of `SCAN` into the function and will thus compile a call to whichever function whose name comes next in the input stream to Flight:

```
c foo
```

2.2 Aliasing

Due to the nature of the kernel's initial assembler, the names of the functions are subject to the limitations of C language identifiers. The resulting names are cumbersome. It is possible to simply compile a call to these functions under a more suitable name, but this introduces the overhead of an extra function call every time the renamed function is called. A better way is to create a name entry which is associated with the address of the code associated with the original name, thus compiling a call to the function's new name generates the same machine code as compiling a call to its original name.

```
: alias c l c SCAN c DEFN_AS CMPRET
```

The function `alias` will look up the address of the next name in the input stream, read in the new name, and make it into a name entry pointing to the address. The names of the kernel functions are then changed where it is convenient:

```
alias   CMPJMP          (JMP)
alias   CMPJMPZERO      (JMP0)
alias   CMPJMPPLUS      (JMP+)
alias   CMPCALL         (CALL)
alias   CMPRET          ;
alias   NUMC            #
alias   NUMI            $n
alias   LOOK            $l
alias   DEFN            $:
alias   SCAN            >$
alias   WRITE1          #>
alias   READ1           >#
alias   TENSTAR         10*
alias   CMPFETCHA       (A@)
alias   CMPSTOREA       (A!)
alias   CMPFETCHAPLUS   (A@+)
alias   CMPSTOREAPLUS   (A!+)
alias   CMPFETCHRPLUS   (R@+)
```

```
alias   CMPSTORERPLUS   (R!+)
alias   CMPXOR          (XOR)
alias   CMPAND          (AND)
alias   CMPNOT          (NOT)
alias   CMPTWOSTAR      (2*)
alias   CMPTWOSLASH     (2/)
alias   CMPPLUS         (+)
alias   CMPPLUSSTAR     (+*)
alias   CMPDUP          (DUP)
alias   CMPDROP         (DROP)
alias   CMPOVER         (OVER)
alias   CMPTOR          (>R)
alias   CMPRFROM        (R>)
alias   CMPTOA          (>A)
alias   CMPAFROM        (A>)
alias   CMPNOP          (NOP)
```

With arbitrary names now possible, some naming conventions can be established:

- Words which compile code inline have their names surrounded by parentheses, with the exceptions of `;` and `#`.
- The use of the input buffer is represented by the use of `$` in a name.
- The use of the data stack is represented by the use of # in a name.
- Moving or copying an item to/from a location is represented by the use of `>` in a name.

2.3 Conditional Jumps

A way of compiling conditional and unconditional jumps to functions is needed to support looping and and tail-call elimination. This is done in the same manner as for compiling calls to functions:

```
: $j c $l c (JMP) ;
: j c >$ c $j ;

: $j0 c $l c (JMP0) ;
: j0 c >$ c $j0 ;

: $j+ c $l c (JMP+) ;
: j+ c >$ c $j+ ;
```

These respectively compile an unconditional jump, jump-on-zero, and jump-on-positive in the same manner previously done for calls.

2.4 Interactive Use

It is necessary to perform manipulation on values while either interacting with Flight or while compiling code. Words for this purpose are created by compiling the code for a machine instruction or Flight word and associating a name to it. Feeding this name to Flight will cause its function to be executed immediately instead of compiled.

```
: @ (>A) (A@) ;
: ! (>A) (A!) ;
: XOR (XOR) ;
: AND (AND) ;
: NOT (NOT) ;
: OR (OVER) (NOT) (AND) (XOR) ;
: 2* (2*) ;
: 2/ (2/) ;
: + (+) ;
: +* (+*) ;
```

```
: DUP (DUP) ;
: DROP (DROP) ;
: OVER (OVER) ;
: NOP (NOP) ;
```

Not all machine instructions and kernel functions shown in subsection 2.2 can be used interactively. The reason for this is that in order to execute a word, its name must first be scanned in, its address looked up, and the code called. This would overwrite values placed by the user in the A register and bury those on the return stack. At worst, this could corrupt the execution of the Flight kernel itself.

2.5 Literals And Inline Compilation

```
: n c SCAN c NUMI CMPRET
```

The function `n` reads in the string form of an unsigned decimal number and converts it to binary form.

```
: #+ c # c (+) ;
: n# c n c # ;
: l# c l c # ;
```

The function `#+` takes a number on the data stack and compiles a literal load instruction for it, followed by an addition instruction. The next two functions compile literal loads from decimal numbers and from looked-up addresses of functions.

```
: negate (NOT) n# 1 (+) ;
: (negate) c (NOT) n# 1 c #+ ;
```

The first function, `negate`, is a straightforward implementation of the negation of a two's-complement binary number. The machine code is only three instructions, which execute in three cycles total and consume only half of a memory word. A function call and return use up a total of four cycles and two memory words. It is thus advantageous to inline the code of `negate` where possible.

The `(negate)` source code looks similar to that of negate, except that it compiles calls to the compiling words instead. When `(negate)` is called, it will compile the code of `negate` inline with existing code at that location. This is used to add the ability to read in a positive number in its negative form:

```
: -n c n (negate) ;
: -$n c $n (negate) ;
: -n# c -n c # ;
```

It is common in code to increment or decrement by a fixed number, and so some inlining words are created for this purpose:

```
: (N-) c -n c #+ ;
: (N+) c n c #+ ;
```

Subtraction is implemented in a similar manner:

```
: - (negate) (+) ;
: (-) c (negate) c (+) ;
```

Finally, inlining versions of other short or common words are created:

```
: (@) c (>A) c (A@) ;
: (!) c (>A) c (A!) ;
: (OR) c (OVER) c (NOT) c (AND) c (XOR) ;
```

2.6 Special Terminal Characters

The current input pre-processor to Flight uses whitespace ASCII symbols as string delimiters. It is impossible to enter those delimiters as strings for storage and later output. Here are some words to output the whitespace characters, with the names based on the C language escape codes:

```
: \a n# 7 n# 1 c #> j #>
: \b n# 8 n# 1 c #> j #>
: \t n# 9 n# 1 c #> j #>
: \n n# 10 n# 1 c #> j #>
: \v n# 11 n# 1 c #> j #>
: \f n# 12 n# 1 c #> j #>
: \r n# 13 n# 1 c #> j #>
: \s n# 32 n# 1 c #> j #>
```

In order, these output the alarm (terminal bell), backspace, horizontal tab, newline, vertical tab, formfeed, carriage return, and space characters. There is currently no way to print them as strings, so the counted string output is done piecewise by outputting the count (1) followed by the character itself. The Flight output post-processor will convert this into a single ASCII character.

2.7 Conditionals

It is possible to write loops and other conditional program flow structures using the words from subsection 2.3, but this is effectively assembly programming and makes the code harder to write and read. The higher-level if...else construct is implemented thus:

```
: if n# 0 c (JMP0) l# HERE_NEXT (@) (N-) 1 ;
: if- n# 0 c (JMP+) l# HERE_NEXT (@) (N-) 1 ;
: else (>R) c NEW_WORD (R>) (>A) (A!) ;
```

The `if` word compiles a jump-on-zero to address zero and leave on the data stack the address at which the zero address value is stored. The `if-` word does the same for a jump-on-positive. Code is then executed as usual until `else`, which aligns the compilation address to the next empty memory word and overwrites with this new address the zero address of the conditional jump instruction compiled by the `if` word. The net effect is to jump over the code compiled between `if` and `else` if the condition tested by the conditional jump instruction is true. Combined with the `j` word, this allows for the construction of any kind of loop, and for the creation of more complex conditionals:

```
: <=
(-) (DUP)
if- (DROP) -n# 1 ;
else
 if n# 0 ;
 else -n# 1 ;

: >=
(-) (DUP)
if- (DROP) n# 0 ;
else
 if n# 0 ;
 else -n# 1 ;
```

These leave a -1 (true) on the data stack if the first number on the data stack is lesser-than-or-equal/greater-than-or-equal than the second number, else they leave a 0 (false). This flag can then be used by an `if...else` construct.

2.8 String Constant Compilation and Output

If a program is to output messages to a user, there must be a way to store strings in the code for later output. The first thing needed is a way to allocate the required space to store the string in the code. This word is also important for lexical closures (section 2.14).

```
: allot (DUP) if c ALIGN (N-) 1 j allot else (DROP) ;
```

A word to copy strings in memory is also required. The addresses are taken from the data stack:

```
: $copy
(>R) (>A)
(A@) (A>) (+) (N+) 1
: $copy-loop (A>) (OVER) (XOR)
if (A@+) (R!+) j $copy-loop
else (DROP)
(R>) (DUP) (>A) (A!) ;
```

This word is then used to copy a string from the input buffer into the code dictionary:

```
: $>c
l# INPUT (@)
l# HERE (@)
(OVER) (@) (N+) 1 c allot
c $copy
c ALIGN
j POP_STRING
```

Similarly, the following word takes the address of a word in the code dictionary and copies it into the input buffer, as if it had originated from the input stream:

```
: c>$
(DUP) (@) c PUSH_STRING
l# INPUT (@)
c $copy ;
```

This word takes the address of a counted string and sends the string to the output stream:

```
: cs>
(DUP) (@) (+) (N+) 1
(A>) (>R)
: cs>-loop (R>) (OVER) (OVER) (XOR)
if (>R) (R@+) c #> j cs>-loop
else (DROP) (DROP) ;
```

The `cs>` word can then used to output the newest string in the input buffer:

```
: $>
l# INPUT (@)
c cs>
j POP_STRING
```

The next two words look up the name of a string and output it. The first does so by copying first to the input buffer while the second outputs directly from the original string contents.

```
: $print c l c c>$ j $>
: csprint c l j cs>
```

Finally, a single character or number from the data stack can be stored into the code dictionary:

```
: #>c l# HERE (@) (!) ;
```

2.9 Memory De-allocation

All the words compiled so far cannot be removed from memory. Their name and code dictionary entries are permanent. To make changes possible, a means of de-allocating this memory is required. The word `forget` takes the name of a function from the input buffer and moves the point of compilation to the beginning of that function's name and code entries, effectively erasing it. Since memory is allocated sequentially, forgetting a function necessarily also forgets any other name and code entries compiled after those of the forgotten function.

The code to do this is fairly complex and is thus built from the bottom up. The first component function returns a 0 if an address on the data stack matches the tail address of the string in the input buffer. This happens if the name of the function to be forgotten is found in the name dictionary.

```
: match?
l# INPUT (@) c STRING_TAIL (XOR) ;
```

The next function tests to see if the end of the name entry dictionary has been reached, meaning that the name of the function to be forgotten has not been found.

```
: end?
l NAME_END @ # (XOR) ;
```

The `erase` function moves the compilation pointers to the beginning of the name and code entries of the forgotten function. The function cannot be looked up anymore, and the next compiled code and the next defined name will overwrite those of the forgotten function.

```
: erase
(DUP) l# THERE (!)
(DUP) (N+) 1 l# INPUT (!)
(@) (DUP)
(N-) 1 l# HERE (!)
l# HERE_NEXT (!)
j ALIGN
```

From these the code for `forget` is built up:

```
: forget
c >$
l# THERE (@) (N+) 1 (DUP)
: forget-loop
l# INPUT (@)
c COMPARE_STRINGS
c match?
if (DROP) c STRING_TAIL (N+) 1 (DUP) c end?
 if (DUP) j forget-loop
 else (DROP) c POP_STRING ;
else c erase (DROP) ;
```

2.10 Elementary Math Functions

These are the building blocks for further mathematical operations. They are currently very limited and some are shown for example only.

2.10.1 Min/Max/Abs

These functions are straightforward. They respectively return the larger (or smaller) of two numbers on the data stack, and the absolute value.

```
: max (OVER) (OVER) (-) if- (>R) (DROP) (R>) ; else (DROP) ;
: min (OVER) (OVER) (-) if- (DROP) ; else (>R) (DROP) (R>) ;
: abs (DUP) if- (negate) ; else ;
```

2.10.2 Unsigned Multiplication

There is no multiplication hardware in the underlying Gullwing machine, therefore integer multiplication is implemented via the common shift-and-add method. However, this algorithm is accelerated by the use of a conditional multiply step instruction, which adds the second number on the data stack to the first only if the first number is odd.

This example function multiplies two 15-bit unsigned numbers into a single unsigned 30-bit unsigned number. The range is limited since the result of a full multiplication must fit in the range of a 32-bit signed number and the product of two unsigned 16-bit numbers could overflow this range[1].

```
: 15x15
(>R)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*)
(R>)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*)
(>R) (DROP) (R>) ;
```

2.10.3 Unsigned Division

The `U/` function implements unsigned integer division by iteratively comparing and subtracting, as in manual long division, and leaves the quotient and the remainder on the data stack. It does not intrinsically handle division-by-zero and so the divisor is checked at the beginning.

The `divby0msg` code stores an error message into memory and provides a name reference to it. The `divby0` function sends that message to the output stream. The `errcontext` redirects the remaining input stream to the output stream, providing a crude way of seeing at which point the division-by-zero error occurred. The `divby0check` function compares the divisor with zero and if true, outputs the error message followed by the context.

```
: divby0msg >$ DIV_BY_0_ERROR $>c
: divby0 l# divby0msg c cs> ;
: errcontext c \s c >$ c $> j errcontext
: divby0check (DUP) if ; else (DROP) (DROP) c divby0 j errcontext

: U/
c divby0check
n# 0 (>A)
: U/-loop
(OVER) (>R) (DUP) (R>) c <=
if
(DUP) (>R) (-) (R>)
(A>) (N+) 1 (>A)
j U/-loop
else
(DROP) (A>) ;
```

[1] There exist means of composing wider multiplications from this, but I have not explored them yet.

2.11 Function Redefinition

A useful property that comes from separating function naming and code compilation is the possibility of redefining a function under the same name[2]. In this case, PUSH_STRING has a design flaw which needs repairing: it does not set the tail of the counted string memory it allocates. This burdens other functions later on. The correct solution is to fix the kernel, but in the meantime a patch will suffice.

The `set_tail` function finds the tail of the topmost string in the input buffer and sets it to contain its own address.

```
: set_tail l# INPUT (@) c STRING_TAIL (DUP) (!) ;
```

The following code begins a new code dictionary entry, compiles code to execute `PUSH_STRING` and `set_tail` in sequence, and associated the name PUSH_STRING with the address of this code. Looking up the address of `PUSH_STRING` will now find this new version.

```
NEW_WORD c PUSH_STRING j set_tail >$ PUSH_STRING DEFN_AS
```

If the naming was done first as usual, the code `c PUSH_STRING` would end up calling to the new instance of PUSH_STRING, creating an endless recursion when executed.

2.12 Binary To Decimal Conversion

The code here performs the inverse of the `NUMI` kernel function: it converts a signed integer into a decimal number string.

The `minus?` function returns 1 if the number is negative, 0 otherwise.

```
: minus?
if- n# 1 ; else n# 0 ;
```

To allocate the correct amount of memory for the decimal string, the number of digits required must be found out in advance. This is done by comparing with increasing powers of ten which fit in 32 bits.

```
: numstrlen
c abs (DUP) (N-) 10
if- (DROP) n# 1 ; else (DUP) (N-) 100
if- (DROP) n# 2 ; else (DUP) (N-) 1000
if- (DROP) n# 3 ; else (DUP) (N-) 10000
if- (DROP) n# 4 ; else (DUP) (N-) 100000
if- (DROP) n# 5 ; else (DUP) (N-) 1000000
if- (DROP) n# 6 ; else (DUP) (N-) 10000000
if- (DROP) n# 7 ; else (DUP) (N-) 100000000
if- (DROP) n# 8 ; else (DUP) (N-) 1000000000
if- (DROP) n# 9 ; else (DROP) n# 10 ;
```

The `#>$` function allocates space for the decimal representation of a number, plus one for a negative sign if needed, and fills that space backwards with the remainders from successive divisions by ten until the remainder is zero.

```
: #>$
(DUP) c numstrlen (OVER) c minus? (+) c PUSH_STRING
(DUP) if- n# 45 l# INPUT (@) (N+) 1 (!) c abs else
(>R) l# INPUT (@) c STRING_TAIL (N-) 1 (R>)
: #>$-loop
n# 10 c U/ (>R) n# 48 (+) (OVER) (!) (N-) 1 (R>)
(DUP) if j #>$-loop else (DROP) (DROP) ;
```

It is now a trivial matter to create a function to directly output the decimal form of a binary number on the data stack:

```
: #$> c #>$ c $> ;
```

[2]It could also be used to create nameless, anonymous functions, but there has not been a need for such yet.

2.13 Function Composition

It is not possible to pass via the data stack an arbitrary number of arguments between functions since the stack is finite in depth and a function cannot determine how much of it is in use at the time. This severely limits function composition. The solution is to place the output of the first function into the input buffer as if it had been read from the input stream, ready to be processed by the second function. The number of arguments that can be passed is then limited only by the amount of free memory. The example shown here composes two function to calculate the mean of an arbitrary segment of the Fibonacci number sequence.

The `1fib` function takes two adjacent Fibonacci numbers and calculates the next number in the sequence, discarding the oldest original number.

```
: 1fib
(OVER) (>R) (+) (R>) ;
```

This function is then placed in a loop which allocates and fills a counted string with Fibonacci numbers[3].

```
: nfibx
c PUSH_STRING l# INPUT (@) (DUP) (>R) c STRING_TAIL (R>) (N+) 1 (>A)
: nfib-loop
(>R) c 1fib (DUP) (A!+) (R>) (DUP) (A>) (XOR)
if j nfib-loop
else (DROP) (DROP) (DROP) ;
```

The mean of these numbers is then the sum of all the elements of the string, divided by its length:

```
: nmean
n# 0
l# INPUT (@) (DUP) (>R) c STRING_TAIL (R>) (N+) 1 (>A)
: nmean-loop
(>R) (A@+) (+) (R>) (DUP) (A>) (XOR)
if j nmean-loop
else (DROP) l# INPUT (@) (@) c U/ c POP_STRING ;
```

The following function composition example leaves on the stack the mean of the first eight Fibonacci numbers:

```
n 1 n 0 n 8 nfibx nmean
```

2.14 Lexical Closures

It is necessary for some functions to have persistent, modifiable values stored within their code, unlike literals and global variables. This is known as a lexical closure, and makes new kinds of functions possible.

The `create` function compiles a literal load and a jump instruction. The literal is the address of the first memory location after the jump, and the address of the jump is set by `does`, in the same manner as for `if...else`, so as to skip over the memory pointed to by the literal. When the code runs, the address of the skipped memory is left on the stack.

```
: create
n# 0 c # l# HERE_NEXT (@) (N-) 1 (>R)
n# 0 c (JMP) l# HERE (@) (N-) 1
l# HERE (@) (R>) (!) ;

alias else does
```

[3] This version terminates when the current storage address matches that of the tail of the string. A decrementing counter version is also possible:

```
: nfibc
(DUP) c PUSH_STRING l# INPUT (@) (N+) 1 (>A)
: nfib-loop
(>R) c 1fib (DUP) (A!+) (R>) (N-) 1 (DUP)
if j nfib-loop
else (DROP) (DROP) (DROP) ;
```

This version uses fewer instructions, but must do a relatively slow subtraction instead of a fast XOR.

A closure can be used to make global variable storage more convenient to manipulate. The `var` function takes a number as an argument, allocates that number of memory locations, and compiles code to return the address of the first of those locations.

```
: var c create (>R) (N-) 1 c allot (R>) c does ;
```

This is used to create global variables. Executing `first` will return the address of the one memory location allocated within it.

```
: first n 1 var ;
```

Storing a number in first and recovering it later for output is straightforward:

```
n 4 first !
first @ #$> \t
```

Closures can also contain executable code. The `EXECUTE` kernel function take the address of the closure and does a function call to it.

```
: pass n# 1 c #$> c \t create n# 2 c #$> c \n ; does n# 3 c #$> c \t c EXECUTE ;
```

The output of `pass` is: `1 3 2`

2.15 Higher-Order Functions

Higher-order functions are functions which can take a function as an argument or return one as a result.

2.15.1 Accumulator Generator

An accumulator generator takes an argument and returns a function which will increment that original argument with its current one.

```
: accgen
c NEW_WORD (>R)
c create (>R) c #>c (R>) c does
c (@) c (+) c (DUP) c (A!) c ;
(R>) ;

: name-as c >$ c DEFN_AS ;

n 3 accgen name-as foo
n 2 accgen name-as bar
n 5 foo #$> \t
n 5 bar #$> \t
n 2 foo #$> \t
n 2 bar #$> \t
n 7 foo #$> \t
n 7 bar #$> \t
n 100 foo #$> \t
n 100 bar #$> \t \n
```

The output of this segment of code is: `8 7 10 9 17 16 117 116`

2.15.2 Fibonacci Generator

The `fibgen1` function is similar in structure as `accgen`, except that it stores two separate values and names the created function before compiling it.

```
: fibgen1
c :
c create (>R) c #>c (R>) c does
c create (>R) c #>c (R>) c does
c (OVER) c (@) c (OVER) c (@) c (+) c (>R)
c (OVER) c (@) c (OVER) c (!) c (DROP)
c (>A) c (R>) c (DUP) c (A!) c ; ;
```

2.15.3 Fibonacci Generator (Optimized)

The code compiled by `fibgen1` amounts to 20 instructions, not including the code for the two lexical closures. Instead of repeating it every time a function is generated, a function call can be compiled to a function that takes addresses as arguments.

```
: memfib
(OVER) (@) (OVER) (@) (+) (>R)
(OVER) (@) (OVER) (!) (DROP)
(>A) (R>) (DUP) (A!) ;

: fibgen2
c :
c create (>R) c #>c (R>) c does
c create (>R) c #>c (R>) c does
l# memfib c (CALL) c ; ;

n 0 n 1 fibgen2 fibonacci
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t
fibonacci #$> \t \n
```

The output is: `1 2 3 5 8 13 21 34`

2.15.4 Map

The map function applies a function to each element of a set. In Flight, this means performing the same operation on each element of a counted string. The example used here is the Caesar cipher, which shifts the letters of a message up or down by a secret amount.

The `caesar` function takes a number and an address from the data stack and adds the number to the contents of the memory location.

```
: caesar
(>R) (>A) (R>) (A@) (+) (A!) ;
```

The `caesargen` function creates an instance of `caesar` with the number to be added as a lexical closure. Using the negative of the original number applies the Caesar cipher in reverse, decoding the message. In this case the secret amount is three.

```
: caesargen
c :
c create (>R) c #>c (R>) c does
c (@) l# caesar c (CALL) c ; ;

n 3 caesargen encode
-n 3 caesargen decode
```

The `map1` function applies a function to each memory location in a counted string.

```
: map1
(DUP) (>R) c STRING_TAIL (R>)
(N+) 1
c l (>R)
: map1-loop
(DUP) (R>) (DUP) (>R) c EXECUTE (N+) 1
(OVER) (OVER) (XOR)
if j map1-loop
else (DROP) (DROP) (R>) (DROP) ;
```

It is now possible to encode and decode strings without having to write loops:

```
>$ ABCD
l INPUT @
DUP cs> \t
DUP map1 encode
DUP cs> \t
DUP map1 decode
$> \t \n
```

The output of this example is: `ABCD  DEFG  ABCD`

2.15.5 Map Generator (Optimized Caesar)

The lexical closure of `caesar` is always a single literal, and the call to caesar is at the tail of the compiled code. This new version of `caesargen` uses these facts to replace the lexical closure with a literal load instruction, and eliminate the tail call with a jump.

```
: caesargen
c : c # l# caesar c (JMP) ;
```

The `map1` function is limited since it can only stride over a counted string with an interval of one memory word, and must repeatedly copy and `EXECUTE` the address of the mapped function. The address of the function to be mapped and the number of memory words it manipulates is known in advance and can be used to create a tailored mapping function which compiles a direct function call and adds the correct memory stride to the address.

```
: mapgen
c (DUP) c (>R) l# STRING_TAIL c (CALL) c (R>)
n 1 # c # c (+)
c NEW_WORD
c (DUP) c l c (CALL) (>R) c # (R>) c (+)
c (OVER) c (OVER) c (XOR)
c if (>R) c (JMP) (R>)
c else c (DROP) c (DROP) c ; ;
```

The new `caesargen` function is used just like the old one:

```
n 5 caesargen encode1
-n 5 caesargen decode1
```

The `mapgen` function is used to compile a mapping version of the encoding and decoding functions. The `cipher` function will encode every second letter only.

```
: cipher n 2 mapgen encode1
: decipher n 1 mapgen decode1
```

The example is as before, except that the use of the map function is now implicit:

```
>$ lmnopq
l INPUT @
DUP cs> \t
DUP cipher
DUP cs> \t
DUP decipher
$> \t \n
```

The output is: `lmnopq  qmsouq  lhnjpl`

2.15.6 Map Generator (Printing Fibonacci Numbers)

The function `print$#` outputs a tab-delimited list of numbers taken from the input buffer.

```
: print# (@) c #$> c \t ;

: print$# n 1 mapgen print#

n 1 n 0 n 8 nfibx l INPUT @ print$# POP_STRING
```

The output is: `1  1  2  3  5  8  13  21`

A Flight Kernel Bug Fixes

No substantial changes to the kernel have been done to implement the extensions, but some bugs and design flaws needed immediate correction.

A.1 ALIGN and NEXT-SLOT

The `ALIGN` function adjusts the `HERE`, `HERE_NEXT`, and `SLOT` kernel variables so as to continue compilation at the next free code dictionary location. The `NEXT_SLOT` function points `SLOT` to the next available instruction slot.

The concept of a null slot, which denotes a memory word in which no instructions had yet been compiled, was introduced to fix a bug which would sometimes leave the zeroth slot of a memory word unused and thus filled with a `PC@` instruction. This would cause the Gullwing processor to skip the remainder of the instructions in that memory location.

A.1.1 FIRST_SLOT

Begin compiling at zeroth slot.

```
LIT [address of SLOT] >A
LIT [mask for slot 0] A! ;
```

A.1.2 LAST_SLOT

Setup at fifth slot, so the next compilation will occur at the zeroth slot.

```
LIT [address of SLOT] >A
LIT [mask for slot 5] A! ;
```

A.1.3 NULL_SLOT

Null slot mask. Denotes that the location pointed to by `HERE` is empty.

```
LIT [address of SLOT] >A
LIT [mask for null slot] A! ;
```

A.1.4 ALIGN

Makes `HERE` point to the next free location so a name entry does not end up pointing at the tail of the previous function or at a literal. Makes `HERE_NEXT` point to the following location. Marks `HERE` as empty with `NULL_SLOT`.

```
LIT [address of HERE_NEXT] >A A@
DUP >R LIT 0 R!+                (zero (PC@) for compilation and increment)
R> A!                           (update HERE_NEXT)
LIT [address of HERE] >A A!     (update HERE)
JMP NULL_SLOT
```

A.1.5 NEXT_SLOT

Point to the zeroth slot if the location pointed to by `HERE` is empty (null slot), else point to the next free slot, else `ALIGN`.

```
LIT [address of SLOT] >A A@                 (get slot mask)
LIT [null slot mask] OVER XOR JMP0 HEREEMPTY
LIT [fifth slot mask] OVER XOR JMP0 HEREFULL
2* 2* 2* 2* 2* A! ;                         (next 5-bit slot)
HEREFULL:
DROP CALL ALIGN JMP FIRST_SLOT
HEREEMPTY:
DROP JMP FIRST_SLOT
```

A.2 COMPILE_OPCODE

Takes an opcode and compiles it at the next empty HERE/SLOT location. Assumes that the current slot is full. Leaves SLOT pointing to the next full slot. This fails for PC@, since its opcode is all zeroes, but it is not necessary to compile it explicitly since ALIGN zeroes memory before compilation.

Calls NEXT_SLOT at the beginning now instead of at the end.

```
call NEXT_SLOT
>R LIT [address of SLOT] >A A@ ~        (get slot mask and invert)
R>                                      (place opcode on top)
OVER OVER & JMP0 COMPILE                (slot 0)
2* 2* 2* 2* 2*                          (shift opcode by one slot)
OVER OVER & JMP0 COMPILE                (slot 1)
2* 2* 2* 2* 2*
OVER OVER & JMP0 COMPILE                (slot 2)
2* 2* 2* 2* 2*
OVER OVER & JMP0 COMPILE                (slot 3)
2* 2* 2* 2* 2*
OVER OVER & JMP0 COMPILE                (slot 4)
2* 2* 2* 2* 2*
COMPILE:                                (then we *have* to be in slot 5)
LIT [address of HERE] >A A@
>A A@ + A! DROP ;                       (compile opcode, drop mask, return)
```

A.2.1 Future Work

The whole slot mask shifting mechanism is wasteful. Instead, COMPILE_OPCODE could shift the contents of HERE and add in the opcode in the fifth slot. The slot mask then devolves into a counter. Since there are less than 32 slots the counter can be a simple one-hot count, which eliminates the need for an addition. However, ALIGN will then have to check and shift the opcodes towards the zeroth sloth if required.

A.3 DEFN

The DEFN function, which takes a name and creates a name dictionary entry with it, has been split into two parts to make function redefinition possible (section 2.15).

A.3.1 NEW_WORD

Returns an aligned address to compile into.

```
call ALIGN
LIT [address of HERE] >A A@ ;
```

A.3.2 DEFN_AS

Takes the aligned address of a code entry (usually from HERE) and the address of a SCANed string (from INPUT), converts it to a name entry, and updates THERE. The string must be the only one in the input buffer as it is converted in place, so INPUT must be pointing to it and its tail must be at THERE. Because of this, INPUT does not need to be changed.

```
LIT [address of THERE] >A A@ >A A! (store address in (THERE))
LIT [address of INPUT] >A A@       (get string start address)
LIT -1 +                           (move to first free location before it)
LIT [address of THERE] >A A! ;
```

A.3.3 DEFN

Defines a name entry for the current code entry location.

```
CALL NEW_WORD
JMP DEFN_AS
```

A.4 PUSH_STRING (Future Work)

The `PUSH_STRING` function should set the tail instead of `SCAN_STRING`. Otherwise, every function afterwards that allocates memory using `PUSH_STRING` has to set the tail, whose value is known the instant it is allocated.

References

[LaF05a] Eric LaForest, *Unit 2-3 (IS 102B) report: The Gullwing stack computer architecture*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.

[LaF05b] ———, *Unit 4-5 (IS 101B) report: The Flight programming language*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.