# Windmill: A Large Code Development System for the Parallax Propeller

## Interpreter Byte Code Specification

The Windmill byte code is meant to normally be executed from very slow media, so a certain amount of processing per code is acceptable compared to, for example, Large Memory Model PASM (LMM) where very fast interpretation tends to be a priority. In Windmill, small code will be faster code just because instruction fetches normally take much longer than execution.

It was decided early on to make Windmill a stack based language like Forth because even with pushing and popping and Hub RAM access, stack access in RAM is much faster than instruction fetching from the external media and it reduces the need for operation tokens to be accompanied by argument tokens locating the data they are to operate on.

It was also a priority to fit as much helper functionality into the interpreter as possible for things like string manipulation, since these functions would be very slow if implemented with looping Windmill token code. Priority was given to functions which are extremely common, such as basic Forth stack manipulation, and functions which involve a lot of looping.

Windmill byte code has two types of token, *Native* and *Helper*. Native tokens are converted directly into synthetic PASM instructions and executed; they are very similar in principle to LMM instructions. Helper tokens trigger more complex functions to be performed by the interpreter, like PASM **CALL**s in LMM.

There is a shared Cog RAM datra structure called the *Instruction Vector* which indexes byte codes to their ultimate functions. For Helper tokens, the S and D fields index setup and execution helper routines; for Native tokens, the I field provides flags which modify how the synthetic PASM instructions are set up.

Note: Arguments following tokens are big-endian since the 3 bit data in an extended token is always the most significant bits of the data. Longs written with store and fetch are little endian like the aligned longs in the stack, though they don't have to be long aligned.

### Native Tokens

Native byte codes are always of the form %00xx_xxxx, so there are 64 of them. The 6 least significant bits of the byte code are inserted into the INSTR field of a standardized instruction via MOVI, and that instruction is then executed. This provides access to most PASM instruction functions with 8 bit instead of 32 bit codes and a very modest amount of Cog RAM wrapper code, leaving plenty of room for Helper code.

What Windmill normally does when it detects a Native token is:

1. Place the token in the INSTR field of the execute instruction. Since MOVI transfers 9 bits including the Z, C, and R bits, which are not included in the token, these three bits are set.
2. Pop the stack into a reserved S long
3. Pop the stack into a reserved D long
4. Clear the C and Z flags and make sure the R bit is set
5. Execute the **`INSTR D,S WC WZ`**
6. Save the flags for later inspection
7. Push the contents of the reserved D long, which now contains the instruction result,

This scheme provides all the support necessary to profitably use about 35 `PASM` opcodes. Some of the problems that arise with other opcodes are:

- Some only take one argument, making the second pop to D redundant
- Some produce no result, making the push of D redundant
- Some serve no purpose unless flags are preserved from a previous instruction
- The R bit is essentially part of the opcode of the Hub RAM access operators, meaning that if R is always set there is no way to set up the Hub RAM **`WR*`** commands which require R=0
- While the S field for **`HUBOP`** could be supplied by pushing an appropriate value, individual **`HUBOP`**s vary as to whether they need two arguments or produce a result, and the **`HUBOP`** S would have to be awkwardly pushed before any actual data argument in D.

This brings us to the *Instruction Vector*. For Native byte codes this list indexes the appropriate *setup modification bits*. These are in the I field, positioned to be rotated out of the vector entry as they are needed.

The meaning of the bits in the I-field of a Native token's IV entry are:

- Bit 31: Rotate this bit into the instruction R field
- Bit 30: 1 = Pop D before execution
- Bit 29: 1 = automatically restore flags

Whether D is pushed after execution would finally depend on how R was set in the instruction. This allows us to accommodate just about everything except hub RAM writes and **`HUBOP`**, and while those could be supported with a more complex bit scheme the number of useful instructions recovered would not justify the extra inner interpreter overhead; the most important ones become helper functions.

With this system an interpreter consisting of about 30 longs of code and a vector that is shared with the helper token interpreter gives us direct access to nearly all `PASM` functions. The few exceptions are:

- Codes $04, $05, $06, and $07 (%100, %101, %110, and %111) are nonfunctional because they are reserved for nonexistent `PASM` operators
- **`JMP/ JMPRET`**, **`DJNZ`**, **`TJNZ`**, and **`TJZ`** serve no function since, as in LMM, Windmill execution branches have to be done by a Helper code which changes the Windmill instruction pointer
- **`WAITVID`** might be implementable but as with Spin not likely to be useful
- The **`CMP`**s are left out in the cold.

- **MOV** is prepped for two pops / push result; since **MOV** pops S, D, then copies S to D before pushing **D**, this has the effect of removing the original second element from the stack. This is a Forth function called **NIP**.

Indexing all of the possible Native tokens would normally require 64 IV elements. On coding the interpreter it became clear that there would only be about 38 normal Helper tokens and 15 extended, leaving a surplus of 12 longs devoted to Native mod bits only. By adding only 2 longs to the inner interpreter it was possible to fold the index into two parallel 32-entry ranges, so that:

- Bits 31, 30, 29: modify opcodes %000000 - %011111
- Bits 27, 26, 25: modify opcodes %100000 - %111111

...for a net saving of 10 longs of interpreter code.

## Helper Tokens

There are two types of Helper token. Tokens of the form %01xx_xxxx are *Normal Helper tokens*. There can be 64 of these. Tokens of the form %1abc_xxxx are *Extended Helper tokens* which contain 3 bytes of argument data; there can be up to 16 of these.

Like Native tokens, Helper tokens index into the IV but instead of flag bits they find two code pointers. The S field points to an appropriate *Prep Handler* which collects any arguments needed by the instruction; this can be one, two, or three pops or a code argument consisting of one, two, or four additional byte tokens. The three abc bits of an extended helper are treated as the most significant bits of any additional code argument data. There are also special Prep functions for relative jumps, all of which take a single argument added to the instruction pointer and some of which pop a condition flag.

All Prep Handlers finish by branching to the D field of the IV entry, which points to the actual function code for the helper. Helper functions always finish by branching to an appropriate *Finish Point* to push any results as appropriate before executing the next token.

Having both prep and finish code shared among helper functions allows the actual individual functions to be quite compact, so we can have more of them.

## Extended Helper Token Functions

Nearly all Windmill tokens work with data on the stack, but there obviously needs to be a way to get data onto the stack in the first place.

**N3** (1 byte, 3 data bits) – Push sign extended 3 bit literal (for 0, 1, -1, etc.)
**N11** (2 bytes, 11 bits) – Push a sign extended 11 bit literal onto the math stack.
**N19** (3 bytes, 19 bits) – Push an unsigned 19 bit positive literal (generally an address) onto the math stack.
`-- N`

**AJMP** (3 bytes, 19 bit target) Absolute jump.

**ACALL** (3 bytes, 19 bit target) Absolute call. The standard way to call a Windmill word.
**BCALL** (3 bytes, 19 bit target) Buffered Absolute call. As **ACALL** but commands the memory server to buffer the word in Hub RAM if this is possible.

**SJMP** (2 bytes, 11 bit target) Short Relative jump.
**SJMPZ** (2 bytes, 11 bit target) Pop the stack and take the SJMP if the result is zero
**SJMPNZ** (2 bytes, 11 bit target) Pop the stack and take the SJMP if the result is not zero
`FLAG --`

**CASE** (2 bytes, 11 bit target) – Pop a compare value from the stack, and if it is not equal to the (unpopped) reference value left on the stack take the relative jump.
`REF VAL -- REF`

**CMP** Construct a compare instruction, pop two arguments off the stack, perform the comparison based on the synthetic `CON` field built from the abc bits, and push the result . Although we have only three bits of data and `CON` is four, by inverting bit `a` to create the an unprovided high `d` bit we can produce the most useful test conditions:

| cond. | flags | d=!a | cba | operator |
|---|---|---|---|---|
| if_a | if_nc_and_nz | 0 | 001 | > |
| if_b | if_c | 1 | 100 | < |
| if_ae | if_nc | 0 | 011 | => |
| if_be | if_c_or_z | 1 | 110 | =< |
| if_eq | if_z | 1 | 010 | == |
| if_ne | if_nz | 0 | 101 | <> |

`N1 N2 -- FLAG`

**@COG** (2 bytes, 9+2 bit target) Read a long of interpreter Cog RAM and push it
`-- N1`
**!COG** (2 bytes, 9+2 bit target) Pop a value and write it to a long of interpreter Cog RAM
`N1 --`

A fair number of seemingly more complex functions are just **@COG** or **!COG** with an appropriate argument, including **rp!**, **rp@**, **sp!**, **sp@**, **rflag**, and **sflag**, and of course all Windmill access to the Special Function Registers **INA**, **DIRA**, etc.

**RET** (1 byte, 3 bit target) pop the R-stack and returns from a `CALL`, releasing 0 to 7 Longs of local stack frame variable space. Larger stack frames must be cleared with `RSTKA`.

**STKA** (1 byte, 3 bit target) Adjust the M-stack pointer by the sign extended number of Longs (-4 to +3). This implements `DROP 4` (negative argument) through `RECOVER 3` (positive arguments.)

**Normal Helper Token Functions**

The only Normal Helper which takes a direct argument is:

**N32** (5 bytes) – Push a 32-bit literal onto the math stack.
**-- N**

All other Normal Helpers are one byte.

Stack-based flow control (note that simple M-stack based **JMP** is a **!COG**):

**CALL** pops a destination from the math stack, pushes the return point onto the R-stack, and jumps.
Addr --

Some primitives lifted directly from Forth:

| | | | | |
|---|---|---|---|---|
| **!** | **@** | **c!** | **c@** | Read & write longs and bytes to memory |
| **MOVE** | **MOVE>** | **FILL** | | Block move and fill (character only) |
| **>r** | **r>** | **r@** | | R-stack manipulation |
| **DROP** | **DUP** | **OVER** | **PICK** | M-stack manipulation |
| **ROT** | **ROLL** | **SWAP** | | |

Unlike Forth, and more like Spin, Windmill uses byte addressing exclusively. All memory operations work in EEPROM, but note that some – especially **MOVE>** which must work down – might be very slow.

A few basic Hub RAM instructions must be supported which can't be constructed by the Native code engine:

| | | | |
|---|---|---|---|
| **WRBYTE** | **WRLONG** | | Differentiated from RD* by different R |
| **COGINIT** | **COGID** | **COGSTOP** | Need special S value, different setups |

Windmill offers unsigned multiplication and division implemented in PASM. The high long of the last multiply can be retrieved via **cog@ Math_Hi**, and the remainder of the last division via **Div_Rem**..

**U*** **U/**

**RSTKA** (2 bytes) Adjust the R-stack pointer
This can be used to reserve and release a local storage stack frame.

**RSTKI** (2 bytes) Push to the M-stack the location (suitable for **!** and **@**) of the R-stack frame local variable at R-stack position 0-127. Note that position 0 is overwritten by further R-stack pushes, including word call returns.

Windmill also offers flexible block and string handlers written in PASM. Windmill block functions specifically target three types of buffer:

- *Stacked Strings* which consist of all the bytes of the string pushed from *end to start* followed by the length. Stacked strings can be longer than 255 bytes.
- *Pascal Strings* which are accessed by memory pointers which point to a byte length, followed by the byte data. They are limited to 255 bytes.
- *Buffers* which are accessed by memory pointers to the first byte of data and for which lengths are specified by other arguments. Buffers can be longer than 255 bytes.

**SJSTR** (2 bytes plus string) Pushes the address of the instruction argument, which is the length of a following string literal *and* the Pascal-style length descriptor for that literal, and execute a jump around the string literal.
`-- addr`

**PJSTR** (2 bytes plus string) Like `SJSTR`, but creates a stacked string by pushing all of the bytes of the string individually from right to left, then pushing the number of bytes in the string, then jumping around the string
`-- bn ... b3 b2 b1 n`

**PUSHSTR** Pop the address of a Pascal String and create a stacked string from it.
`addr -- bn ... b3 b2 b1 n`

**PUSHBUF** Pop the length and address of a buffer and create a stacked string from it. `PUSHBUF` does push the length, but it is not limited to 255 bytes.
`addr len -- bn ... b3 b2 b1 n`

**POPSTR** Pop an address, and then unstack a stacked Pascal string at that address.
`bn .. b3 b2 b1 n addr --`

**POPBUF** Pop an address, then a length, then that many bytes and write the bytes at the address. Unlike `POPSTR`, `POPBUF` *does not write the length byte, and the length is not limited to 255 bytes.*
`bn .. b3 b2 b1 n addr --`

**INBUF** Take a character, the address of a buffer, and the length of the buffer and return the byte position within the buffer where the character is found, or -1 if it does not occur, leaving the buffer parameters on the stack.
`n addr char – n addr pos`

**INSTR** Take a character the address of a Pascal string and return the byte position within the string where the character is found, or -1 if it does not occur, leaving the string address on the stack.
`addr char – addr pos`

**CMPSTR** takes the same parameters as `POPSTR` but pushes true or false according to whether

the stacked string is the same as the existing Pascal string. Leaves both the stacked string and string address on the stack.

```
bn .. b3 b2 b1 n addr  -- bn .. b3 b2 b1 n addr FLAG
```

**CMPBUF** takes the same parameters as `POPBUF` but pushes true or false according to whether the stacked string is the same as the existing memory buffer. Leaves both the stacked string and buffer address on the stack. (The difference between `CMPSTR` and `CMPBUF` is that `CMPSTR` includes the Pascal string length byte in the comparison.)

```
bn .. b3 b2 b1 n addr  -- bn .. b3 b2 b1 n addr FLAG
```

**STRRANGE** Process a stacked string to perform an action or adjustment on each byte that is inside or outside of a specified range.

The action argument has adjustment in bits 0..7 and the following bit fields:

bit 10: 0=act inside range 1=act outside range
bit 9: 0=add adjustment 1=replace with adjustment
bit 8: 0=act writes byte 1=act suppresses byte

`STRRANGE` is meant to automate functions like `UCASE` / `LCASE` and `ASCII` / numeric conversion.

```
bn .. b3 b2 b1 n min max act – bn .. b3 b2 b1 n
```

**RSTKA** (2 bytes, sign extended 8 bit target) Adjust the R-stack pointer by adding the signed byte argument number of longs -128 to +127 to it, generally to create or release a stack frame).

**RSTKI** (2 bytes, unsigned 8 bit target) Push onto the M-stack a relative pointer via the R-stack pointer by 1 to 255 Longs (generally to index into a stack frame). Note that the argument is subtracted from the R-stack pointer, so that if a stack frame was created via `RSTKA` with a positive argument N, `RSTKI` arguments 1 thru N will access the reserved Longs. Further pushes to the R-stack (including word call return addresses) will overwrite the Long at index 0.