Propeller Assembly Language

What is Propeller Assembly?

Propeller Assembly is the low-level 32-bit instruction set designed specifically for the Propeller chip. Though most of this language's instructions will be familiar to experienced assembly programmers, there are many that are specific to the Propeller's multicore architecture and a few features are unique to the language.

What is noteworthy about Propeller Assembly?

Propeller Assembly includes instructions specific to its multicore architecture, including those for cog/hub interaction, lock bits (semaphores) for managing shared memory blocks, and cog counter and video generator hardware.

Propeller Assembly has additional features worth noting:

- `IF_X` (Conditions): every Propeller Assembly instruction has an optional "condition" field that dynamically determines whether or not it executes at run time based on current `C` and `Z` flag states. This feature simplifies critical timing routines; enabling multi-decision code to execute along the same path, taking the same amount of time, regardless of the decision.
- `WC`, `WZ`, `NR`, `WR`: every Propeller Assembly instruction has an optional "effect" field that enables the writing or no writing of the `C` and `Z` flags or destination register upon instruction execution. This combined with `IF_X` enables single-path, multi-decision code that is easy to time with certainty.
- `JMPRET`: a mechanism to "call" other routines and eventually return to the instruction that follows the `JMPRET` without the use of a call stack. This allows for efficient use of memory and implementation of simple task-switching within a cog.

See the complete Assembly Opcode Table

Where does Assembly code live and where does it get processed?

Propeller Assembly code exists in the Propeller Application image that is stored in Main RAM during runtime, but it is executed by copying it to Cog RAM and processing it directly from there.

For the initial startup, the Propeller chip runs its Boot Up Procedure and the Spin Interpreter (stored in Main ROM) is copied to Cog 0's RAM. This Spin Interpreter fetches chunks of the application code, called tokens, from the Main RAM and executes it.

When that code is a `COGNEW`(*AsmAddress*…) or `COGINIT`(*AsmAddress*…) command, the Propeller starts the designated cog to load its Cog RAM with 496 consecutive longs of data from Main RAM, starting at the *AsmAddress* location. (This overwrites any code or data that may have been in the target cog's RAM). The cog then executes the assembly code directly from its Cog RAM, beginning at address 0.

How do I indicate the end of Assembly code?

Most assembly programs consist of infinite loops that don't need any end-of-code indicator. In the rare case where you need to make an assembly program terminate at the end of its operation, you need to instruct it to shut down the cog that is running it. You can do this with the Propeller Assembly `COGID` and `COGSTOP` instructions. It may look something like this (assuming `MyID` is an available register):

```
        cogid    MyID    'get the ID of the cog I'm running in
        cogstop  MyID    'use the ID to terminate myself
```

If you do not do this, you may run into strange behavior, and here's why: When an Assembly routine is launched, the cog fills its RAM with 496 longs from main memory, beginning at the *AsmAddress* specified in the COGNEW or COGINIT command. If your routine is less than 496 longs, the cog will also be loading whatever is in adjacent memory, which could be data, variables, or even another Assembly program from the same DAT block. If your Assembly program is not an endless loop, and it does not terminate by identifying its cog and shutting itself down, the cog will continue to try and execute whatever follows the logical end of code. This usually results in undesirable, and sometimes unpredictable, behavior.

What happens when execution reaches the end of a Propeller Assembly program?

Unlike with Spin, the compiler has no way to tell where the end of a Propeller Assembly program is, so it is up to the developer to end it appropriately.

If execution reaches the end of Propeller Assembly, and there is no code there to cause either cog termination or endless looping, the cog will continue execution past the end of code and into the next registers. What follows code is typically variables and data, and following that is unknown data that just happened to follow the assembly in the application image- either way, the cog will try to execute it and that will likely cause bad behavior that may be unpredictable.

Always end your Propeller Assembly programs with either an unconditional jump (to loop the program endlessly)...

```
DAT
  {Assembly code here}
  ...
  jmp #<main_loop_label>                    'Continue back at main asm loop
```

... or with an unconditional jump to itself (a tight endless loop)...

```
DAT
   {Assembly code here}
   ...
   jmp #$                                   'Jump to here, endlessly
```

... or with a COGID/COGSTOP pair of instructions to terminate the cog:

```
DAT
   {Assembly code here}
   ...
   cogid ID                                 'Get our cog ID
   cogstop ID                               'Terminate ourself
ID    long    0                           'Holds Cog ID
```

What's the difference between calling code and launching code?

Calling a piece of code (a spin method or assembly label) executes that code within the current cog. Launching code (a spin method or assembly program) executes that code in a different cog; to execute code in parallel.

Can Spin call Propeller Assembly code?

No, it can launch Propeller Assembly, but can not call it directly since it wouldn't be practical to do so. By design, Spin and Propeller Assembly code execute in separate cogs, so just like

coordinating between two or more cogs, Spin and Propeller Assembly code can communicate with each other via shared memory, if necessary.

Can Propeller Assembly call Spin code?

No, and it wouldn't be practical to do so. By design, Propeller Assembly and Spin code execute in separate cogs, so just like coordinating between two or more cogs, Propeller Assembly and Spin code can communicate with each other via shared memory, if necessary.

Can Propeller Assembly start a new cog?

Yes, using the assembly version of `COGINIT` and specifying a Main RAM target address of the code to launch. You can optionally specify a target cog by ID, or use the lowest-numbered available cog by default. The most practical and recommended way to start a cog is through Spin, however.

How long does it take to start a cog?

It depends on the type of code that is starting the cog and what type of code it is launching into the cog. By nature of design, the most common case is running Spin code that launches other Spin or Assembly code.

Cog Start Time			
Running Code	Launching Code	Clock Cycles[1]	Time @ 80 MHz
Spin	Spin	24,992[2]	≈ 300 µs
Spin	Assembly	8,918[2]	≈ 100 µs
Assembly	Spin	8,400 [3]	≈ 100 µs
Assembly	Assembly	8,196[4]	≈ 100 µs

Note 1: Includes `COGNEW`/`COGINIT` overhead.

Note 2: Minimum cycles in Spin; increases with inclusion of parameters and complexity of expressions and constant values.

Note 3: This is an estimate; launching Spin from Assembly can be tedious and is not recommended.

Note 4: Minimum cycles in Assembly; increases by up to 29 cycles for hub sync of both calling cog and new cog (to execute `COGINIT` and read first long of code from Main RAM).

How can Propeller Assembly access a Spin object's global variables, DAT symbols, or other shared memory?

The first step is to pass the address of the desired memory (global variable, `DAT` symbol, or otherwise) to the Propeller Assembly program. Usually this is achieved by placing the address in the *Parameter* part of the `COGNEW`/`COGINIT` command that launches the assembly cog. Then the assembly program moves the address (which is in the `PAR` register) to another register for reference and possible indexing, and uses the read/write main memory instructions (`RDBYTE`, `RDWORD`, `RDLONG`, `WRBYTE`, `WRWORD`, `WRLONG`) to access the shared memory.

Any changes to the shared memory by either the Spin object or the Propeller Assembly program is immediately viewable by the other.

Do DAT block symbols exist in Main RAM or in Cog RAM?

DAT block symbols exist in Main RAM but also in Cog RAM if they are launched with Propeller Assembly.

The DAT block itself is stored in the application image in Main RAM. Spin-based references to DAT symbols are accessing the corresponding location and data in Main RAM.

When a cog is launched with assembly code, any DAT symbols within 496 longs after the launch point are copied into Cog RAM. Unlike with Spin code, Propeller Assembly code that references those symbols is not accessing the corresponding location and data in Main RAM, but rather it is accessing the corresponding Cog RAM instead; its local copy. In addition, those symbolic references are always to longs of Cog RAM memory, regardless of how the symbol was actually declared.

How are symbols in the DAT block treated in relation to Spin and Propeller Assembly?

The DAT block's primary purpose is to hold fixed data and Propeller Assembly code for the application. Symbolic declarations can be included and used to reference this data and code.

There's nothing preventing the contents of DAT from being modified at runtime, however, which naturally leads to its use to hold symbols as "special" variables. Their special attributes relate to how they are stored in the application and how they are treated by Spin and when launching Propeller Assembly code.

DAT blocks are stored in the application image in Main RAM. Just like code (PUB and PRI), there is only one instance of each DAT block, regardless of how many instances of the containing object there are. This means that Spin-based references to DAT symbols are accessing the corresponding location and data in Main RAM, and it's the same regardless of which instance of that object is making the reference. This is handy to share memory between multiple instances of an object.

When a cog is launched with assembly code, any DAT symbols within 496 longs after the launch point are copied into Cog RAM. Unlike with Spin code, Propeller Assembly code that references those symbols is not accessing the corresponding location and data in Main RAM, but rather it is accessing the corresponding Cog RAM instead; its local copy. In addition, those symbolic references are always to longs of Cog RAM memory, regardless of how the symbol was actually declared. In Propeller Assembly, no Main RAM references can be made by simply using the declared symbolic name; instead, the address of that symbol must be passed from the Spin object and used along with instructions like RDLONG and WRLONG.

How do I determine the execution time of a portion of code?

This is quite easy for Propeller Assembly, but more elusive for Spin code. There are three techniques for timing code with definitive results, each with its "ideal" application.

1. Counting and adding up determinant instruction times (Propeller Assembly only).
2. Using the System Counter (Spin and Propeller Assembly).
3. Toggling a pin and measuring the pulse-width (Spin or Propeller Assembly).

For details, see the Code Execution Time article.

Does Propeller Assembly use a call stack?

No, but it does provide an alternative.

Propeller Assembly language does not implement a call stack since doing so would unnecessarily consume valuable memory and impose undue limitations on applications. Instead, Propeller Assembly provides a different mechanism through the JMPRET instruction (and also CALL, JMP,

and RET) to maintain nested call history. This method requires more developer influence but allows the memory to be used more efficiently (optimizing for the specific application) and has a distinct advantage allowing the implementation of simple task-switching code in real-time systems using JMPRET.