PPCForth

Larry Battraw

v0.42b 05/31/2000

Introduction and Philosophy

PPCForth is a small, rather loosely characterized version of forth. It has a number of standard words, such as `@`, `!`, `emit`, `key`, etc., but does not even come close to implementing the core wordset for an ANS-type forth. As such, I expect it will be roundly ignored by those who consider the ANS standard to be a holy grail of sorts. It includes a few small extensions to the language that I consider quite nice, but will again, doubtlessly annoy savants at large. I hate to say this, but forth as a language is pretty unfriendly. It often seems like the names for things were chosen arbitrarily, and made as cryptic as possible. I still don't fully grasp many tenets of the language, simply because it always seemed like it was written with that purpose.

I wrote PPCForth because there wasn't a single forth I could find which was written for the PowerPC™ in an embedded environment. Every FORTH out was for a different processor, or was simply enormous in complexity and size (C-based forths in particular). PPCForth currently occupies a little over 28K, and has all the functions I have felt necessary to effectively code small utilities, although I am sure people will remind me of things I have forgotten.

I apologize for any bit rot that may have or is taking place. I am working on several projects and use PPCForth as the basis for some of them; this unfortunately means I rarely get the time to do frequent updates.

One small extension to FORTH I have made is simply an attempt to make the raw language a little more palatable to read. In my opinion, looking at the source for a FORTH program always reminds me of what you might see if you attempted to read a document off a disk with errors (abort, retry, fail!). The extension consists simply of the ability to defer execution of word(s), based on parentheses. This makes it a little clearer what a word needs to have sitting on the stack when it is called. An example may make things clearer:

```
 : myword
    emit(12)        \ clear screen
    if(key?) ." A key was waiting" cr then
    .(1)            \ print 1,2,3
    .(2)
    .(3)
    do(10 0) .(i) cr loop
;
```

Yes, I have include a `puts` function, not a `TYPE`. And `s"` only returns an address. Likewise for `gets`. It tickles me pink to be able to declare a variable like `my_str`, and then do: `puts(my_str)`, or something like `if(key?)`, or even `until(0=)`.

A second extension is intended to eliminate the need to worry about which number base you are currently in when compiling, etc. This also can help eliminate those subtle errors where the BASE may be changed somewhere, suddenly causing numbers to either be interpreted incorrectly or instigating false errors (like a hex number while in base 10). To force a number to either hex or binary, prefix it with the following: `0x` or `$` for hex, and `%` or `#` for binary. The `0x` is obviously an offshoot of C, which the `$`, `#`, and `%` are from various assembler dialects. Currently there is a bug related to this; if a string is entered which has valid hex characters in all but the first character (and you're set for the hex base) it will silently accept it, dropping the first invalid character.

Perhaps the most useful part of PPCForth is its ability to act as a small monitor-type program `;-)` It will accept S-records directly from the command line, dump blocks of memory, jump to an absolute address, and is fully ROM-able. It keeps a rom-based portion of its dictionary in one location, and you tell it where to keep all words and data defined thereafter. You will need to edit the source to adjust the I/O-dependent words (if you're not using a PPC403), as well as pointing it to the right place in memory. I currently use `0x00000000` as my base for the system (in the assembler source, although execution actually occurs beginning at (`0xfff80000`) and `0x7fe74000` for the second dictionary and data. My system stack is positioned at `0x7fe7fffc` (contained in `r1`). Check out the source for more specific info on which registers are used for what. There is a large block comment at the top explaining some of the nuances of the system.

Read This Section Before You Run PPCForth

There are two entry points for the system, based on whether you are doing a warm or cold entry. A cold entry initializes everything, while a warm entry preserves the contents of the RAM-based second dictionary. I set it up that way because my RAM is non-volatile, and thus will maintain the contents of my second dictionary just fine. If you don't have NVRAM, you will want to leave things as they are (unless "nvram" is defined), where it does a cold start every time. Otherwise the first time you type in a word that's not defined (or a number), the system will lock or crash! Neat, huh?

The source for PPCForth is written in a dialect understood by the inimitable AS by Alfred Arnold (alfred@ccac.rwth-aachen.de). This assembler will compile code for virtually every processor known to man (and then a few), simply by telling it within the source file which one you want. It even allows you to switch processor types within a single file (although that may not be really useful). I make heavy use of macros, so please do yourself a favor and get this assembler– it really is nice. It runs on Ms-dog (including Windoze 98/nt), OS2, and Linux. The Ms-dog version also has a variant which runs under a DOS extender for DPMI. However, only the *nix version is currently being maintained, to the best of my knowledge.

Strings

Currently PPCForth has minimal support for strings, in the tradition of all "bad" forths. To declare a string variable, simply do the following:

```
 variable my string
<size/4 in bytes> allot
```

i.e. To allocate an 80 byte string called mystr:

```
 variable mystr
20 allot
```

This of course declares a variable called `my_str`, and then allocates a certain number of words (4 byte integers). So if, you wanted a string variable with 16 bytes of space, you would allot 4 words.

String constants are handled by `s"`, which will allocate the size of your string in the current word being compiled, and then return that address (relative to `r28`). To maintain 4-byte alignment, if you declare a string that has a length+1 that is not a multiple of 4, it will pad in 0's. Example:

```
 s" a string"
```

Here, the string appears to have eight characters (we don't count the leading space from `s"`), but that doesn't include the null that is automatically added. So the total is 9 bytes, which is then padded to 12.

Unlike other forths, PPCForth allows and even encourages using the string constants as variables which are automatically initialized. The second dictionary is always in RAM, where new definitions are stored, and so this works every time. You simply use `s"` to declare a string of sufficent length to hold future strings, and save the pointer it returns for use later on in your definition. Kind of like a local variable.

As of this writing I have included `strcpy`, `strlen`, and `strcmp`, as well as `cstrcmp` (counted-strcmp) and `cstrcpy`. These make managing strings a little easier....

'Gotchas'

All languages have their own quirks and pitfalls (Gotchas) which inevitably trip up the beginner. If you somehow managed to skip the paragraph labeled "READ THIS...", go back and read it now.

Any other nasty things you run into are probably unintentional and should be reported, please...

Details, details....

To be able to make use of PPCForth, you will need some type of PowerPC processor. If you're lucky, you already have one, and you're REALLY lucky you're using what I have (a 403GCX). To really make things work without a struggle, you will need some type of 40x-family processor. I would love to support the 8xx series, but not having one, it makes it difficult. The biggest problem with a different PPC type is the serial I/O and multi-tasking code. The serial I/O code is interrupt-driven for receiving characters, which allows you to reset or break out of programs. To use plain polling, modify the 'gets' routine to call 'igetchw' or supply your own code.

Anatomy of a FORTH Word (Internal view)

Within the forth dictionary, words are stored in a particular fashion I found most convenient ;-) There are two dictionaries, each of which is relative to a particular point in memory. If a search of the first dictionary fails to yield a particular word, the second is queried. The second is always held in RAM, while the first may reside in ROM or RAM.

Structure of a Word:

```
 <32-bit value representing total length of word>
<32-bit value representing combined length of word's name and PFA>
<null terminated ASCII string of word's name; padded to 4-alignment>
<PFA- parameter field; contains flags specifying the word's type>
<data or code field; contains the executable code or data of the word>
```

If you're doing multitasking and would like to customize the callback, preempt, or priority values, they are appended to the end of a word like so:

```
 <callback time in clock ticks>
<preempt time in clock ticks>
<priority (lower number==greater priority)>
```

Note that you must define your word and then comma in the values (i.e. they can't be within the definition!). More on this later...

Adding the second 32-bit value to the address of the first will yield the address of the code/data field. With this address, you may subtract four to obtain the address of the PFA. And finally, you may (obviously) add the first 32-bit value to its address to get the address of the word following it. The end of both dictionaries is indicated by a single 32-bit value, equal to zero.

Wordlist and Explanations

All numbers are 32 bit integers, addresses must be aligned on a 4-byte boundary! Only data addresses are relative– code is absolute.

Standard Words

- `\`

starts a one line comment which continues to the end of the line.

```
{words etc.} \ my comment is here
```

- `emit`

prints a single character off stack

```
<char> emit
```

- `key`

accepts a single character, placing the value on stack

```
key <char>
```

- `key?`

returns a boolean value based on whether or not a key is available (-1 for true, 0 for false)

```
key? <flag>
```

- **`cls`**

clears screen (emits an ASCII 12)

- **`puts`**

prints a null-terminated relative string to the console

```
<string addr> puts
```

- **`gets`**

gets a null-terminated relative string from the console

```
<string addr> gets
```

- **`+`**, **`-`**, **`*`**, **`/`**

aritmetic functions

- **`and`**, **`or`**, **`nor`**, **`not`**

logic functions

- **`c@`**

retrieves a byte from an relative address in memory (doesn't need to be 4-byte aligned)

```
<addr> c@ <byte>
```

- **`c!`**

stores a byte at the address specified

```
<byte> <addr> c!
```

- **`ac@`**, **`ac!`**

identical to c!, c@ except they use absolute addresses

- `@`

retrieves a 32-bit word from an relative address in memory (must be 4-byte aligned)

```
<addr> c@ <byte>
```

- `!`

stores a 32-bit word at an relative address in memory (must be 4-byte aligned)

```
<val32> <addr> c@ <byte>
```

- `a@` , `a!`

identical to @ and ! except they use absolute addresses

- `>abs`

returns the absolute address of an address relative to the base of the forth system

```
<rel_addr> >abs <abs_addr>
```

- `<abs`

returns the relative form of an absolute address

```
<abs_addr> <abs <rel_addr>
```

- `swap`

swaps top two items on stack

```
<val1> <val2> swap <val2> <val1>
```

- `over`

copies the item under the top item on the stack to the top

```
<val1> <val2> over <val1> <val2> <val1>
```

- **`>R`**

places the current value on stack on the return/loop stack; Do NOT allow >R or R> to cross the boundaries of a loop or bad things will happen!

- **`R>`**

pulls the top value off the return/loop stack and places it on the stack

- **`strlen`**

returns a count (in bytes) of a null-terminated string. The null is not included in the count.

```
<string addr> strlen <count>
```

- **`cstrcpy`**

copies a counted string to another destination

```
<len> <src addr> <dest addr> cstrcpy
```

- **`strcpy`**

identical to above, but without any count (string must be null- terminated)

```
<src addr> <dest addr> strcpy
```

- **`cstrcmp`**

compares two strings, with the specified length (count). Please check before calling cstr-cmp that the string are of equal length! Otherwise there is a chance that they may be equivalent for the specified length, but no further. For example, " `thi` " and " `this` ", with a length of 3 will count as equal.

```
<len> <str1 addr> <str2 addr> cstrcmp <flag 0/-1>
```

- **`strcmp`**

identical to above, but without a count; this avoids the problem mentioned above.

```
<str1 addr> <str2 addr> strcmp <flag 0/-1>
```

- `1-` , `2-` , `4-` , `1+` , `2+` , `4+`

abrieviated add/subtract functions

```
<val> 2+ <val+2>
```

- `>`

greater than test

```
 <val1> <val2>  >  {-1/ffffffff if val1 is greater than val2;
0 otherwise}
```

- `<` , `=` , `0=`

other comparision tests

- `!=`

this does a equivalence compare, and then NOTs the result. Thus, if you had 0 and 1 on the stack, doing a != results in fffffff being returned instead of 0.

```
<val> <val2> !=  <inverted comparison result>
```

- `cr`

prints a CR/LF

- `dup`

duplicates top item on stack

```
<val> dup <val> <val>
```

- `drop`

drops top item on stack

```
<val> drop
```

- **`sbuf`**

a builtin string variable of 160 bytes in length; it is used by b2asc, so if you use it, be careful that you don't try and print a number with something saved here.

- **`base`**

a builtin variable. Holds the current base for numeric string input and output.

- **`b2asc`**

converts a number to an ASCII string and returns a pointer to that string.

```
<number> <string addr> b2asc <string addr>
```

- **`asc2b`**

converts a string to a number. Please note that if you use this function directly there is currently no way to find out if it converted all digits successfully; any errors are reported silently and are invisible from the user standpoint. If I can find a way I like to fix this, I will do it. This problem does not affect numbers entered through interpretation/compilation, which will correctly detect any illegal characters.

- **`EXTENSION`**

I have included an extension in this routine which automatically (and temporarily) changes the base based on a number's prefix. To convert as a hex number prefix the number with `0x` or `$` (such as `0x1fd` or `$1fd`). To convert as a binary number, prefix with `%` or `#` (i.e. `#10101` or `%10101`)

```
<string addr> asc2b <number>
```

- **`hex decimal binary`**

simple functions to set the base. Equivalent to doing

```
a <number> base !
```

- `.`

prints and removes top item on stack in hex

```
123f . 123f {printed}
```

- `,`

used during compilation to directly enter data into dictionary

```
5223 , {places 5223 as a 32-bit value into dictionary}
```

- **`pfa!`**

this is a context insensitive way to alter a word's pfa (parameter field). This allows words to be declared as a particular type, such as INLINE, COMP, XCOMP, etc.

```
<pfa type> word_addr pfa!
```

- **`pfa!`**

this is a context sensitive way to alter the most recently declared word's pfa (parameter field). This allows new words to be declared as a particular type, such as INLINE, COMP, XCOMP, etc.

```
<pfa type> pfa!
```

- **`IMM`** , **`COMP`** , **`XCOMP`** , **`NOCOMP`** , **`VAR`** , **`CONST`** , **`INLINE`** , **`MULTI`** , **`MTRT`** , **`MTFR`**

pfa flag bits which may be OR'd together. They perform the following functions:

  - **`IMM`**

standard and default pfa given to new words; this means they may be compiled into other words and are called in the normal way.

  - **`COMP`**

this type means the word is only allowed in compile mode (from a colon definition).

- `XCOMP`

this type means a word is only allowed in compile mode and will execute when referenced while compiling.

- `NOCOMP`

this type prohibits a word from being referenced/used while compiling.

- `VAR`

this type declares a word as a variable and will return the address of the data (or code) field of the word instead of executing it.

- `CONST`

this type declares a word as containing a single 32-bit value in the data field. Referencing this word will return that value.

- `INLINE`

this type declares a word as containing a raw chunk of data (usually a small section of machine code). Referencing this word will copy all of the data field of a word.

- `MULTI`

this type declares a word as containing trailer information relevant to multitasking.

- `MTRT`

this type declares a word (in addition to the MULTI type) capable of receiving messages for real-time response to stimuli. This also requires the trailer info.

- `MTFR`

this type declares a word (in addition to the MULTI type) to be called at a regular intervals, specified by the callback value. This also requires the trailer info.

- **`exec`**

executes the dictionary entry indicated; note that the address
is the absolute address of the dictionary entry, not the code field address.

```
<entry_addr> exec
```

- ### `create`

creates a dictionary entry of the specified name. After doing this you may ALLOT space or COMMA (,) in values directly. This word takes a string following the word, unlike other words which have values preceding them.

```
create <name>
```

- ### `:` (colon)

(colon) begins the definition of a new word and turns on compilation. All words/constants following this are compiled in, not executed immediately. Semi-colon is used end a new word.

```
: <new_word>
```

- ### `;` (semi-colon)

(semi-color) ends the definition of a new word.

```
: <new_word>  {words}  ;
```

- ### `dolit`

compiles in the code to push a number on the stack.

```
<number> dolit
```

- ### `abort`

called whenever there is an error. It resets all stacks and jumps back to the main loop (effectively a warm start). It attempts to print a string pointed to by `r12` (usually points at the `TIB`), with a lenth stored in `r13`. This is handy if during execution or compilation it encounters something it doesn't like. It can then print: `"problem_string": huh?` For other types of errors, you will simply get the error, followed by a carriage return, and then a `:huh?`.

- **`if else then`**

conditional code exexcution controls. Takes standard FORTH structure of `<flag> if {code} else {code} then`

- **`i`, `j`**

meta-variables which are set by the entry into a begin-until loop; they only return their current value, not an address. `I` is the base, which will count up to the upper limit (`J`).

- **`do loop`**

looping construct. Loops a set number of times, set by two numbers placed on the stack before entering the loop (upper limit and base). The meta-variables I and J contain the values passed to the do-loop and I (the base) will follow the count up to the upper limit.

```
 <upper_limit> <base> do {code} loop
```

- **`begin until`**

Conditional loop construct. Loops until a boolean flag of true is passed to UNTIL.

```
 begin {code} <flag> until
```

- **`leave`**

this word will immediately exit a loop (either a do-loop or begin-until). If loops are nested, it will exit only the inner- most loop level that it is placed at.

```
 10 0 do i . cr
if(key?) leave then
loop
```

- **`ping1`**

simply prints a ' `*` ' (for debugging).

- **`ping2`**

simply prints a ' `!` ' (for debugging).

- `words`

prints a list of all the words in the dictionary. No CR/LFs are printed, so hopefully your term program will wrap to the next line instead of printing only the first line of words.

- `.s`

prints contents of the stack. This can be used to check for under or overflows of the stack (other words never check for an empty stack before pulling off a word).

- `depth`

returns the number of items on the stack (not including the number pushed on indicating count of items on stack).

- `'` (tick)

(tick) returns the dictionary entry address for a word. Can be passed directly to EXEC. Note that the name of the word is passed after the tick, not before it.

```
' <name_of_word>   <address_of_word>
```

- `variable`

creates an entry in the dictionary of the specified name and allocates a single 32-bit word. The PFA (parameter-field-address) is given the type of variable as well, indicating that instead of executing the definition, its code field address should be returned.

```
variable <name_of_variable>
```

- `constant`

creates an entry in the dictionary of the specified name and allocates a single 32-bit word and then stores the value on the stack in that location. When the word is executed it will return that value.

```
<constant value> constant <name_of_constant>
```

- `allot`

allocates N words of space to a recently created word or variable. Please remember that we are allocating 32-bit words, NOT bytes. This word does not perform a sanity check to see if there is a currently "open" entry that can have space allocated, so be care- ful!

- `string`

this word copies in a string, terminated by a quote (") to the word being compiled currently. It will then put code in place to return the address of the string on the stack when the word being compiled is executed. Not used directly in programs usually. The address returned is relative.

```
string <string>"   <address of string>
```

- `s"`

this word is an alias for the above word (string) and so functions identicly. Use this word instead of string. Note that there must be a space between the s" and the string, although the terminating quote needs no space. This word can only be used during compilation of a new word.

```
s" <string>"  <addr>
```

- `."`

this word prints a string, terminated by a quote. Simply a combination of s", followed by puts. Only usable during compilation.

```
." <string>"    {prints the string}
```

- `c_colon`

meta-variable which returns the address of the next point in memory that data/code may be placed during compilation.

- `t_colon`

meta-variable which returns the address of the top of the current colon definition being compiled.

- `[ ]`

these words toggle compilation off and back on, allowing code to be executed during compilation.

```
 : myword [ 1 1 + . ] {code} ;
```

This will print the number two and then continue compiling.

- **`forget`**

this word will eliminate the indicated dictionary entry and all entries AFTER it.

```
 forget <name_of_word>
```

- **`dump`**

prints a hex and ASCII listing of a section of memory. Uses an absolute address. To view a dump of word, you might do the following:

```
 ' <word>  \ get the absolute address
dump      \ print dump of word
```

- **`jump`**

executes a routine, stored at the indicated absolute address.
Note that this differs from EXEC in taking the actual address of the code to execute (doesn't need a dictionary entry), as opposed to the address of a dictionary entry (word).
If the code jumped to has a blr opcode (powerpc mnemonic) at its end, execution will be returned to the FORTH system, provided the LR (link register) has not been altered.

```
 <addr_to_jump_to> jump
```

- **`go`**

jumps to address defined in local.inc as '`loadpt`'. Handy if you load something and jump to it over and over.

- **`char`**

returns the ASCII value of the character following the char word, seperated by a space. I.E. char `*` will return a `42` (0x2a hex)

- `cold`

performs a cold restart which initializes all stacks and zaps the second dictionary (initializes it as empty).

- `warm`

performs a warm restart. Does exactly the same thing as a cold restart without killing the second dictionary.

- `dict1@`

returns the absolute address of the base of dictionary one (considered to be in ROM).

- `dict1!`

changes the base address of dictionary one; not advisable!

- `dict2@`

returns absolute address of the base of dictionary two (considered to be in RAM).

- `dict2!`

changes the base address of dictionary two; not advisable!

- `dict2_eof`

returns absolute address of the end of dictionary two

- `b_of_rom`

returns absolute address of the base of the forth system in ROM (equivalent to `r27`)

- `b_of_ram`

returns absolute address of the base of the forth system in RAM (equivalent to `r28`)

- **`<S Record>`**

Any time an s-record is entered, it will place that data in memory immediately. Currently only 32-bit address records are handled properly (`S3` type), as well as the `S0` (header) and `S7` (end of record) types.

PPC403-Specific Words

- **`tblo`**

returns value of system time base, low 32 bits

- **`tbhi`**

returns value of system time base, high 32 bits

- **`pit@`**

returns value of the count-down programmable interval timer (32-bit)

- **`pit!`**

sets the value of the count-down programmable interval timer; at start up, the pit will automatically begin counting down the value loaded to 0 and stop. It counts at the same rate as the system clock and time base (25,000,000 counts per second)

- **`tcr@`**

returns the value of the TCR special-purpose register (timer control register)

- **`tcr!`**

sets the value of the TCR special-purpose register

- **`tsr@`**

returns the value of the TSR special-purpose register (timer status register)

- **`tsr!`**

sets the value of the TSR special-purpose register

- **`msr@`**

returns the value of the MSR special-purpose register (machine status register)

- **`msr!`**

sets the value of the MSR special-purpose register

- **`srr0/srr1 @/!`**

sets or returns the save-restore registers

- **`int_entry`**

an inline word which saves registers r3-r9, and the CR

- **`int_exit`**

an inline word which restores the above registers an does an rfi

- **`pit_vec!`**

a word which copies the code necessary to call a PPCForth word from a PIT interrupt; to change the base of the vector table look in the file PPCForth.inc for the equ called evector. Used like so:

```
' my_word pit_vec!
```

- **`led!`**

a very specific word for storing a value to a location in memory (`0x70110000`). Not too useful for other people. (ed.: toggles an LED on the author's specific board.)

ROM Monitor Words

- **`memcpy`**

copies N bytes of memory from one absolute address to another len source destination memcpy

- **`flashc!`**

stores one byte in the FLASH RAM; this uses an algorithm know to work with AMD 29F series devices. PLEASE note that it takes a RELATIVE destination to base of the flash, 0 being the lowest address. It returns a flag indicating success/failure (0==ok).

```
byte rel_addr flashc! <error flag>
```

- **`flcpy`**

copies N bytes of memory into the FLASH RAM. See above; also takes a relative destination. For the current flash device I use (29F040), the sectors are 64K in size and bad things happen if you don't complete a full 64K write to the device (it seems to get confused and doesn't write the proper data).

```
len source rel_dest_addr flcpy <error flag>
```

- **`flash_sector_erase`**

erases one sector of the FLASH RAM; in AMD devices this is 64K, Atmel devices may use 128-byte sectors– but they don't need to be erased in the first place, so ignore this word if you're using Atmel.

```
rel_addr flash_sector_erase <error flag>
```

- **`reflash`**

Using the AMD flash algorithm, it copies the data at the address " `rombase` " is set to (64K) to the first sector of the flash. Thus, if you recompile PPCForth and upload it to that location you can make the changes permanent with this command.

Atmel-Specific Flash Words

- **`atmel_fwrite`**

This performs the same function as `flcpy`, but for Atmel devices. Just to make it difficult, it taskes an ABSOLUTE source address rather than a relative one like `flcpy`. The Atmel devices I've used expect sectors to be written in 128 byte increments, so keep this in mind.

```
len_in_bytes abs_source rel_dest atmel_fwrite
```

- **`at_unprotect`**

Some PROM burners set the security bit, making it possible to erase the device, but not to write to it! This is bad. This word will remove that protection so that it is possible to rewrite it in-circuit. You only need to execute it once unless you use a PROM burner on the chip again.

- **`turbo`**

simple word to toggle the bit in the `IOCR` (`0x4000`) on to allow turbo mode in the 403GCX devices

- **`slow`**

toggles above bit off

Multi-tasking Words

Before I launch in the the list, I might add that although I do use the term "real time" or "rt", this may not be strictly in accordance with what other people might call "real time". What I mean in is that the words (in the FORTH sense of the term) that make use of these services may be called at fixed regular intervals with precedence over "non-rt" words, or called when a message is sent to them, suspending normal execution of other words of lesser priority or marked "non-rt".

- **`taskman`**, **`rttaskman`**

these words represent the multitasking managers and are NEVER executed directly

- **`sendmsg`**

sends a message to another task. If the task is defined as real-time (`MTRT`), it immediately causes the destination to be executed. Typical latency on a 403GCX running at 33 MHz is 22 microseconds. Execution will be deferred if a task of greater priority is scheduled. Sendmsg returns a flag indicating success/fail. The values are as follows:

- `0` success
- `-1` destination word not found
- `-2` destination word's queue is full
- `-3` destination word's queue is in use by another task.

Commands less than or equal to " `sighi` " (currently `9`) will be passed to the task manager and are used to stop/start/etc the dest task (value is ignored).

```
command value destination sendmsg <flag returned>
```

- **`getmsg`**

gets a message directed to the currently executing task. If a message is not present in the que, `0xffffffff` will be returned. getmsg destination source value command

- **`addtask`**

adds the indicated word to the task que and schedules its execution if it is either a MTFR or simple MULTI word. If the word does not have the MULTI type in the PFA set, it will be assigned default values for priority (`0x10`) and preeempt (`0x1000`). Adding a task that does not exist could have bad consequences; it is rather intolerant of invalid addresses. Addtask returns the absolute address the the entry within the task manager's task list. To create a word that has preset parameters for priority, preempt, and callback it is necessary to comma in the values representing the desired settings. For example, for a callback of 10000, a preempt of 500, and a priority of 10 you would do something along the lines of:

```
 : myword
begin
<blah blah blah>
1 0= until
;
10000 , 500 , 10 ,
' myword make-multi
' myword make-fr
' myword addtask
word_addr addtask header_address
```