Symmetric Multiprocessing in zeptoforth on the Raspberry Pi Pico

Getting zeptoforth

- zeptoforth can be gotten from:
- https://github.com/tabemann/zeptoforth
- The most recent release at the time of writing, 0.26.4, can be gotten from:
- https://github.com/tabemann/zeptoforth/releases/tag/v0.26.4
- Note that this release is necessary for some of the code examples to work properly.
- The code examples in this presentation are at:
- https://github.com/tabemann/zeptoforth/tree/master/test/rp2040/present

Multiprocessing and Multitasking

- zeptoforth, aside from in kernel-only builds, is a preemptive multitasking system with priority scheduling. Its scheduler also attempts to balance the time allocated to each task within a given priority.
- On the Raspberry Pi Pico and other compatible RP2040-based systems zeptoforth also has support for symmetric multiprocessing combined with multitasking.
- The only exceptions to this are that attempting to write to flash once the second core is booted will result in undefined behavior, and zeptoforth can only be rebooted from the first core.

A Basic Demo

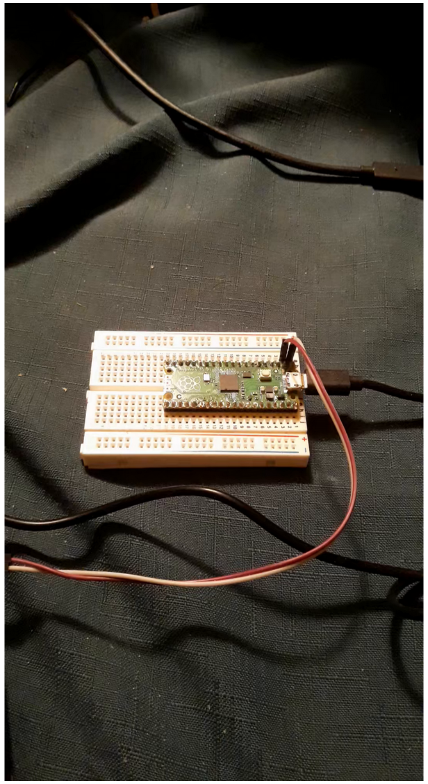
- Before we get into things, a basic demo program for zeptoforth on the Raspberry Pi Pico:

```
task import

led import

: init-test ( -- )

    0 [: begin led-toggle 750 ms again ;] 320 128 512 1 spawn-on-core run

    150 ms 0 [: begin led-toggle 600 ms again ;] 320 128 512 0 spawn-on-core run ;
```

- This toggles the LED attached to the MCU at two different rates simultaneously from two different cores.

A Basic Demo (cont'd)

- Some video of the previous:



Basic Multitasking Words

- These words are in the `forth` module:
- `pause ( -- )`
- This relinquishes control of the current core by the current task to the next task that is ready to execute on it.
- `ms ( ms -- )`
- This temporarily halts the execution of the current task for the specified number of milliseconds.
- `user ( "name" -- )`
- This defines a task-local variable by the given name. Note that it is only applied to tasks created after it is defined.

Basic Multitasking Words (cont'd)

- `cpu-count ( -- cpus )`
- This returns the number of cores supported on the MCU.
- `cpu-index ( -- index )`
- This returns the index of the core on which the current task is executing.
- These words are in the `task` module:
- `current-task ( -- task )`
- This returns the current task.

Basic Multitasking Words (cont'd)

- `spawn ( xn … x0 count xt dsize ssize rsize -- task )`
- This spawns a new task on the same core as the spawning task.
- `spawn-on-core ( xn … x0 count xt dsize ssize rssize core -- task )`
- This spawns a new task on an arbitrary core.
- `run ( task – )`
- This starts the execution of a task.
- Note that tasks are allocated high in RAM and subtract space from the main task's dictionary without impacting its `here` pointer.

Task Creation (cont'd)

- An example of task creation on the second core of the RP2040:

```
task import
: init-test ( -- )
    0 [: begin ." *" 1000 ms again ;] 320 128 512 1 spawn-on-core run ;
```

- This creates and starts a task on the second core which prints an asterisk once every second which has a dictionary of 320 bytes (including user variables), a data stack of 128 bytes, and a return stack of 512 bytes (including nested interrupts and multitasker state).

Priority

- Each task is assigned a priority, with higher priorities being assigned a higher value and lower priorities being assigned a lower value. Valid priorities range from -32768 to 32767. Tasks default to priority 0. Note that except when resolving priority inversion, only tasks of the highest priority that can run will run.
- These words are in the `task` module:
- `task-priority! ( priority task -- )`
- This sets a task's priority.
- `task-priority@ ( task -- priority )`
- This gets a task's priority.

Priority (cont'd)

- The following code illustrates the use of priorities:

```
task import
: init-test ( -- )
    0 [: begin ." *" 250 ms again ;]
    320 128 512 spawn 0 over task-priority! run
    0 [: begin 10 0 ?do ." x" 100000 0 ?do loop loop 4000 ms again ;]
    320 128 512 spawn 1 over task-priority! run ;
```

- This prints alternating sequences of '*' and 'x'; when the higher priority task is printing 'x's, the lower priority task which prints '*'s cannot execute.
- Note that priorities operate on a per core-basis.

Timeouts

- Blocking operations in zeptoforth have support for timeouts. Timeouts by default are off, but they may be enabled as follows:
- These words are in the `task` module:
- `timeout`
- This is a user variable which specifies the current timeout from the beginning of each blocking operation in 100 µs increments.
- `no-timeout`
- This value stands for no timeout when set as `timeout`.
- `x-timed-out`
- This is an exception raised when a timeout is reached.

Timeouts (cont'd)

- The following code illustrates the use of timeouts with rendezvous channels (which will be mentioned later):

```
task import

fchan import

fchan-size buffer: my-fchan

: init-test ( -- )

    my-fchan init-fchan

    0 [: begin [: 10000 timeout ! my-fchan recv-fchan ;] extract-allot-cell . again ;]

    320 128 512 0 spawn-on-core run ;
```

- The displays the message "block timed out" after one second passes.

Semaphores

- Semaphores in zeptoforth are the usual multitasking construct allowing multiple tasks to wait on events signaled by any number of tasks.
- These words are in the `sema` module:
- `sema-size ( -- bytes)`
- This returns the size of a semaphore in bytes.
- `init-sema ( limit counter addr -- )`
- This initializes a semaphore at address *addr* with initial counter *counter* and maximum counter limit *limit*; `no-sema-limit` specifies an unlimited semaphore.

Semaphores (cont'd)

- `take ( sema -- )`
- This decrements the counter of semaphore *sema* and if the counter is less than zero, block the current task until the counter becomes zero.
- `give ( sema -- )`
- This increments the counter of semaphore *sema* and if the counter is less than or equal to zero afterwards, unblock the longest-blocked task waiting on the semaphore.

Semaphores (cont'd)

- The following code illustrates the use of semaphores across cores to print an asterisk every second:

```
task import

sema import

sema-size buffer: my-sema

: init-test ( -- )

    1 0 my-sema init-sema \ This is a binary semaphore, as indicated by the limit of 1

    0 [: begin my-sema take ." *" again ;] 320 128 512 0 spawn-on-core run

    0 [: begin my-sema give 1000 ms again ;] 320 128 512 1 spawn-on-core run ;
```

Notifications

- Notifications in zeptoforth are a lightweight data transfer/synchronization construct; they differ from semaphores in having the advantages of that they are capable of higher performance than semaphores and they store single cells of data in mailboxes (with a limit of 32 mailboxes per task) that can be updated when tasks are notified, while having the disadvantages of that only a single task can be notified at a time and that potential wakeups can be lost if multiple notifications arrive at a task in close succession or while a task is not waiting for any notifications.

Notifications (cont'd)

- These words are in module `task`:
- `config-notify ( mailbox-addr mailbox-count task -- )`
- This initializes notifications for a specified task, including both the size of the task's mailbox space in cells and its address.
- `wait-notify ( mailbox -- x )`
- This waits for a notification on a specified mailbox of the current task and then, once the notification is received or if a notification had been received already, returns the contents that mailbox.

Notifications (cont'd)

- `notify ( mailbox task -- )`
- This notifies a task on a mailbox without updating the mailbox.
- `notify-set ( x mailbox task -- )`
- This notifies a task on a mailbox while setting the mailbox to a fixed value.
- `notify-update ( xt mailbox task -- )`
- This notifies a task on a mailbox while using an *xt* to mutate the value of the mailbox. Note that said *xt* should not carry out any operations that affect the state of the multitasker, including any IO.

Notifications (cont'd)

- `mailbox@ ( mailbox task -- x )`
- This gets the value of a mailbox on a task without blocking.
- `mailbox! ( x mailbox task -- )`
- This sets the value of a mailbox on a task without notifying the task.

Notifications (cont'd)

- The following code illustrates the use of notifications across cores to count each second:

```
task import
1 constant notify-count
notify-count cells buffer: notify-area
variable notified-task
: init-test ( -- )
     0 [: begin 0 wait-notify . again ;] 320 128 512 0 spawn-on-core notified-task !
     notify-area notify-count notified-task @ config-notify -1 0 notified-task @ mailbox!
     notified-task @ run
     0 [: begin ['] 1+ 0 notified-task @ notify-update 1000 ms again ;]
     320 128 512 1 spawn-on-core run ;
```

Locks

- Locks in zeptoforth are self-explanatory. They are non-recursive (so do not lock a lock you have already locked), and they have support for priority inheritance to resolve priority inversion.
- These words are in the `lock` module:
- `lock-size ( -- bytes )`
- This is the size of a lock in bytes.
- `init-lock ( addr -- )`
- This initializes a lock at address *addr*.

Locks (cont'd)

- `lock ( lock -- )`
- This claims *lock* if it is not already locked, and if it is, block the current task until all previously-locking tasks have released *lock* and then claim *lock*.
- `unlock ( lock -- )`
- This releases *lock*, which has been claimed by the current task.
- `with-lock ( xt lock -- )`
- This claims *lock*, blocking if necessary, execute *xt*, and then release *lock*; if an uncaught exception occurs in *xt*, release *lock* and then re-raise the exception.

Locks (cont'd)

- The following code illustrates the use of locks to protect IO across cores:

```
task import

lock import

lock-size buffer: my-lock

: init-test ( -- )
    my-lock init-lock
    0 [: begin [: $100 0 ?do i h.2 space loop ;] my-lock with-lock again ;]
    320 128 512 0 spawn-on-core run
    0 [: begin [: 100 0 ?do i . loop ;] my-lock with-lock again ;]
    320 128 512 1 spawn-on-core run ;
```

Queue Channels

- Queue channels in zeptoforth are simple queues consisting of a fixed number of fixed-sized slots for messages that are shared between tasks. Sending to a full queue channel will block the current task until another task receives at least one message from the queue channel, and receiving from an empty queue channel will block the current task until another task sends at least message to the queue channel.
- Note that larger queue channels can achieve significantly greater bandwidth than smaller queue channels, at the expense of increased latency.

Queue Channels (cont'd)

- These words are in the `chan` module:
  - `chan-size ( bytes count -- bytes )`
- This is the size of a queue channel that may contain up to *count* messages of size *bytes*.
  - `init-chan ( bytes count addr -- )`
- This initializes a queue channel at address *addr* that may contain up to *count* messages of size *bytes*.

Queue Channels (cont'd)

- `send-chan ( addr bytes chan -- )`
- This sends a message at address *addr* consisting of size *bytes* to queue channel *chan*; the message will be zero-filled or truncated depending on the queue channel element size. If the queue channel is full, block the current task until it is not full.
- `recv-chan ( addr bytes chan -- recv-bytes )`
- This receives a message into a buffer at address *addr* of size *bytes* from queue channel *chan* and return either *bytes* or the element size depending on which is smaller, truncating the message if necessary. If the queue channel is empty, block the current task until it is not empty.

Queue Channels (cont'd)

- The following code illustrates the use of queue channels across cores to count each second:

```
task import
chan import
64 constant element-count
cell element-count chan-size buffer: my-chan
: init-test ( -- )
    cell element-count my-chan init-chan
    0 [: begin [: my-chan recv-chan ;] extract-allot-cell . again ;]
    320 128 512 0 spawn-on-core run
    0 [: 0 begin dup [: my-chan send-chan ;] provide-allot-cell 1+ 1000 ms again ;]
    320 128 512 1 spawn-on-core run ;
```

Rendezvous Channels

- Rendezvous channels (aka "fchans") in zeptoforth operate much like queue channels with an element count of 1, except that there is no buffer storing queued data but rather data is transferred directly from the sending task to the receiving task, and the sending task must wait for a receiving task to receive on the rendezvous channel before it may continue executing. Additionally, there are no predetermined limits aside from RAM available on the size of data sent via rendezvous channels, as any message up to the size of the receiving task's buffer can be sent.

Rendezvous Channels (cont'd)

- These words are in the `fchan` module:
- `fchan-size ( -- bytes )`
- This is the size of a rendezvous channel in bytes.
- `init-fchan ( addr -- )`
- This initializes a rendezvous channel at address *addr*.
- `send-fchan ( addr bytes fchan -- )`
- This sends a message at address *addr* consisting of size *bytes* to rendezvous channel *fchan*. If there is no task waiting on the rendezvous channel for a message, block until a task receives from the rendezvous channel.

Rendezvous Channels (cont'd)

- `recv-fchan ( addr bytes fchan -- recv-bytes )`
- This receives a message into a buffer at address *addr* of size *bytes* from rendezvous channel *fchan* and return either *bytes* or the sending task's buffer size depending on which is smaller, truncating or zero-filling the message if necessary. If no task is waiting on the rendezvous channel to send a message, block until a task sends on the rendezvous channel.

Rendezvous Channels (cont'd)

- The following code illustrates the use of rendezvous channels across cores to count each second:

```
task import

fchan import

fchan-size buffer: my-fchan

: init-test ( -- )
    my-fchan init-fchan
    0 [: begin [: my-fchan recv-fchan ;] extract-allot-cell . again ;]
    320 128 512 0 spawn-on-core run
    0 [: 0 begin dup [: my-fchan send-fchan ;] provide-allot-cell 1+ 1000 ms again ;]
    320 128 512 1 spawn-on-core run ;
```

Bidirectional Channels

- Bidirectional channels (aka "rchans") in zeptoforth operate much like rendezvous channels, except that the receiving task replies to the message it receives, sending data back to the original sending task. The original sending task blocks until it receives its reply, and the original receiving task blocks until it has sent its reply to the original sending task.
- Bidirectional channels achieve better performance in sending data than using a pair of rendezvous channels to achieve the same.

Bidirectional Channels (cont'd)

- These words are in the `rchan` module:
- `rchan-size ( -- bytes )`
- This is the size of a bidirectional channel in bytes.
- `init-rchan ( addr -- )`
- This initializes a bidirectional channel at address *addr*.
- `send-rchan ( saddr sbytes raddr rbytes rchan -- rbytes' )`
- This sends a message at address *saddr* consisting of size *sbytes* to rendezvous channel *rchan*. Block until a reply is generated, which is copied into a buffer at address *raddr* of size *rbytes*, and return the size of the reply in bytes.

Bidirectional Channels (cont'd)

- `recv-rchan ( addr bytes rchan -- recv-bytes )`
- This receives a message into a buffer at address *addr* of size *bytes* from bidirectional channel *rchan* and return either *bytes* or the sending task's buffer size depending on which is smaller, truncating or zero-filling the message if necessary. If no task is waiting on the bidirectional channel to send a message, block until a task sends on the bidirectional channel.
- `reply-rchan ( addr bytes rchan -- )`
- This replies to a message received from a bidirectional channel *rchan* with a message in a buffer at address *addr* of size *bytes*.

Bidirectional Channels (cont'd)

- The following code illustrates the use of bidirectional channels across cores to count each second:

```
task import
rchan import
rchan-size buffer: my-rchan
: init-test ( -- )
    my-rchan init-rchan
    0 [: begin cell [: dup cell my-rchan recv-rchan my-rchan reply-rchan ;]
    with-aligned-allot again ;] 320 128 512 0 spawn-on-core run
    0 [: 0 begin dup [: [: my-rchan send-rchan ;] extract-allot-cell . ;] provide-allot-cell
  • 1+ 1000 ms again ;]
    320 128 512 1 spawn-on-core run ;
```

Streams

- Streams in zeptoforth operate much like queue channels, except that they are designed for sending and receiving arbitrary number of bytes at a time rather than discrete messages.
- Sending and receiving multiple bytes at a time with a stream garners higher bandwidth and lower latency than sending and receiving single bytes at a time with either a stream or a queue channel.

Streams (cont'd)

- These words are in the `stream` module:
- `stream-size ( dbytes -- bytes )`
- This is the size of a stream of size *dbytes* in bytes.
- `init-stream ( dbytes addr -- )`
- This initializes a stream at address *addr* of size *dbytes*.
- `send-stream ( addr bytes stream -- )`
- This sends data to a stream *stream* from a buffer at address *addr* of size *bytes*. The data will be sent as one fixed block, so this will block until enough room is available in the stream to fit all of the data being sent.

Streams (cont'd)

- `recv-stream ( addr bytes stream -- recv-bytes )`
- This receives data into a buffer at address *addr* of size *bytes* from stream *stream* and return the size of the data received in bytes. If the stream is empty, block until the stream contains data.

Streams (cont'd)

- The following code illustrates the use of streams to transfer sequences of bytes between cores:

```
task import

stream import

256 constant my-bytes

my-bytes stream-size buffer: my-stream

: init-test ( -- )

     my-bytes my-stream init-stream

     0 [: begin 16 [: dup 16 my-stream recv-stream type ;] with-allot again ;]

  •  320 128 512 0 spawn-on-core run

     0 [: begin s" 0123456789ABCDEF" my-stream send-stream again ;]

     320 128 512 1 spawn-on-core run ;
```