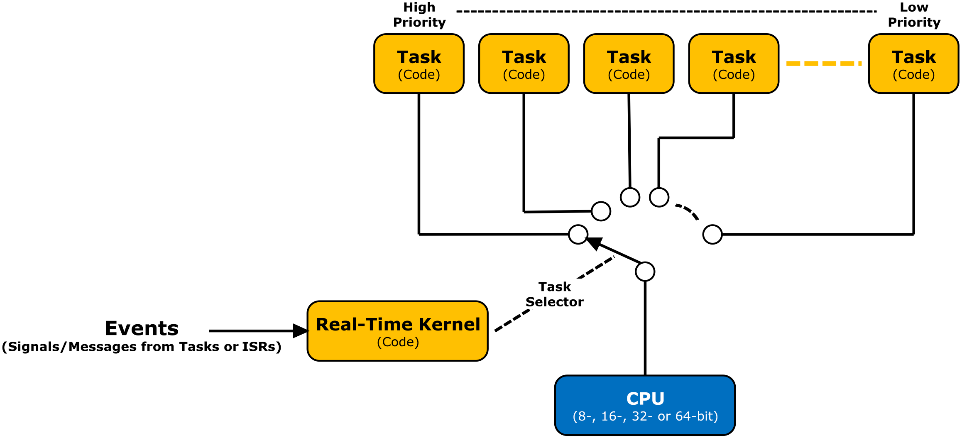
**50**

HBO ICT

TI

*RTOS Reader*

C++ Programming & Software Engineering 1



**TICT-V2CPSE1-15**

Studiejaar 2017-2018

|  |  |
| --- | --- |
|  |  |
| Cursuseigenaar | **Wouter van Ooijen** |
| **Auteur(s)** | **Wouter van Ooijen** |
| **Datum** | **2017-09-04** |
| **Versie** | 1.0 |
|  | |

© Institute for ICT, Hogeschool Uterecht, 2017

Contents

[2. Inleiding 3](#_Toc492285209)

[3. Blink a LED and read a pin 4](#_Toc492285210)

[4. Coroutines 7](#_Toc492285211)

[5. Rtos 9](#_Toc492285212)

[5.1. Create tasks 9](#_Toc492285213)

[5.2. Clocks and timers, and waiting for multiple evenst 10](#_Toc492285214)

[5.3. Event flags and external interfaces 12](#_Toc492285215)

[5.4. Pools and mutexes 14](#_Toc492285216)

[5.4.1. Example 15](#_Toc492285217)

[5.5. Channels 16](#_Toc492285218)

[5.5.1. Example 16](#_Toc492285219)

[6. The NEC IR protocol 18](#_Toc492285220)

[6.1. Protocol descriptipon 18](#_Toc492285221)

[6.2. A design of an IR receiver application 19](#_Toc492285222)

# Introduction

This reader explains some practical aspects of the use of the RTOS and its relation with class diagrams and STDs. It is part of the material fort he second-year course V2CPSE1. The full sources can be found in the v2cspse1-examples repositiry that is used in this course. The last section is the context for an assignment.

# Blink a LED and read a pin



|  |
| --- |
| Subjects |
| * Work while waiting * State machines (FSMs) |

Consider an application that must do two things:

* Blink a LED,
* Read two input pins an copy the AND of the two values to an output pin.

Both parts are very simple to write when it is the ony thing to do.

|  |  |
| --- | --- |
| for(;;){  led.set( 1 );  wait\_us( 50’000 );  led.set( 0 );  wait\_us( 50’000 );  } | for(;;){  out.set( a.get() && b.get() );  } |
| Blinking and copying by two separate loops | |

But how can these two things be combined? One way is to notice that the blink loop spends most of its time in wait\_us(). In this function there will be some loop that just loops around until the requested amount of time has elapsed. A simple way to do this is to ask some OS service for the current time, add the requested wait time, and then loop, wasting time, until that moment has arrived.

|  |
| --- |
| void wait\_us( long long int t ){  auto end = hwlib::now\_us() + t;  while( hwlib::now\_us() < end ){  // waste time  }  } |
| A wait implementation using now\_us() |

When we put the single line that implements the AND inside the while loop, that line will be run as a kind of ‘background task’ whenever the main application (the LED blinker) is idle, without the need to change that main application.

|  |
| --- |
| void wait\_us( long long int t ){  auto end = hwlib::now\_us() + t;  while( hwlib::now\_us() < end ){  out.set( a.get() && b.get() );  }  } |
| Doing background work while waiting |

This trick worked because the blink task spends most of its time waiting, and the AND task doesn’t need to wait. In most cases things aren’t that simple. When we must for instance blink two LEDs at different frequencies we can’t use this trick directly, because both tasks contain wait calls.

|  |  |
| --- | --- |
| // blink at 10 Hz  for(;;){  led1.set( 1 );  wait( 50 ms );  led2.set( 0 );  wait( 50 ms );  } | // blink at 3 Hz  for(;;){  led2.set( 1 );  wait( 165 ms );  led2.set( 0 );  wait( 165 ms );  } |
| Blink at two different frequencies. |  |

To combine these two blinkers in a similar way, we must rewrite the loop as a function that doen’t call wait (it must start, do its business, and return almost immediately), but is called repeatedly. The next function does this by knowing when the LED was last toggeled. When enough time has elapsed since that moment, the LED is toggeled again and the last-toggle-time is updated. Note that the information that must be preserved between calls to the led\_update() function must be stored in global variables, because (normal) local variables are not preserved from one call of the function to the next. (An alternative is to use static local variables, which have local visibility but global lifetime.)

|  |
| --- |
| **pin\_out \* led;**  **long long int last\_toggle = 0;**  **bool level = 0;**  **void led\_update(){**  **auto t = hwlib::now\_us();**  **if( t - last\_toggle > 50'000 ){**  **level = ! level;**  **led->set( level );**  **last\_toggle = t;**  **}**  **}** |
| **Blinking using an update function** |

This style of programming is called a (Finite) State Machine, because it has a state (for the blinker: the LED level and the last toggle moment) and each call of the function (possibly) updates the state. In C++ we can rewrite the led updater to a class that encapsulates the global variables, and provides an update() interface. This makes it possible to create a bunch of such state machine objects (LED blinkers or other types), create an array of pointers to them, and have them ‘serviced’ by the main() without the need for that main to know anything specific about the individual state machines. Since we expect the FSMs to use the current time, we might as well have the main() retrieve it and pass it to the FSMs. This isolates the FSMs from how the current time is obtained.

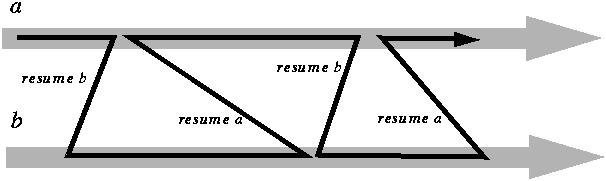
|  |
| --- |
| **class fsm {**  **public:**  **virtual void update( const long long now ){}**  **};** |
| **An interface for a simple FSM** |

|  |
| --- |
| **class blinker : public fsm {**  **private:**  **hwlib::pin\_out & led;**  **long long interval, last\_toggle;**  **bool led\_state;**    **public:**  **blinker( hwlib::pin\_out & led, long long interval ):**  **led( led ), interval( interval ),**  **last\_toggle( 0 ), led\_state( false )**  **{}**  **void update( const long long now ) override {**  **if( now - last\_toggle > interval ){**  **led.set( led\_state = ! led\_state );**  **last\_toggle = now;**  **}**  **}**  **};** |
| **A blinker FSM that implements the simple FSM interface** |

|  |
| --- |
| **auto b1 = blinker( pin1, 100 );**  **auto b2 = blinker( pin2, 39 );**  **fsm \* fsms[ &b1, &b2, ... ];**  **for(;;){**  **auto now = hwlib::now\_us();**  **for( f : fsms ){**  **f->update();**  **}**  **}** |
| **A main loop that services the ‘registered’ FSMs** |

State machines are a lightweight way to do more than one thing ‘at the same time’, but it requires an ‘inside-out’ way of writing that is often more difficult than the natural way (with function calls in the code that can wait).

# Coroutines



|  |
| --- |
| Subjects |
| * Coroutines * Context switching implementation |

The most natural way to blink two LEDs is to write 2 mains.

|  |  |
| --- | --- |
| // blink at 10 Hz  for(;;){  led1.set( 1 );  wait\_us( 50’000 );  led2.set( 0 );  wait\_us( 50’000 );  } | // blink at 33 kHz  for(;;){  led2.set( 1 );  wait\_us( 15 );  led2.set( 0 );  wait\_us( 15 );  } |
| Blinking two LEDs by two for loops | |

A symmetrical way to run these two mains (without having to rewrite one or both as a state machine) is to arrange for the wait\_us() call to continue the execution of the other main, at the place where that main had arrived. If we (the wait call in the first main) continue execution we check wether the requested wait time has elapsed. If so we retrun, if not we (again) activate the other main.

|  |
| --- |
| **void wait\_us( int t ){**  **int start = now\_us();**  **while( start + t < now\_us() ){**  **switch\_to( other\_thread );**  **}**  **}** |
| **A wait call that resumes another part of the application while waiting** |

This requres that we can write a switch\_to function that the CPU starts executing from one main, and when the CPU is finished with that function it emerges in the other main. This can’t be done in pure C/C++, but it can be done with a little assembler.

To return from a function, we must be able to get the CPU in exactly the state it would have when it exited that function in the normal way. For a Cortex M0 CPU this means that the general-purpose registers R4 .. R12 and the special registers SP and PC must have the value they would have when the function exited in the normal way, and the stack (where the SP points to) must contain the ‘old’ values. (The content of the registers R0 .. R3 and the status flags is undefined after a function call, so we don’t need to save or restore those registers.) To be able to do this, we must save those registers (and not overwite the stack).

The obvious place to save the register values is on the (old) stack. Next the old SP is saved in some place where we can find it back, the SP is set to the stack of the other main, and the registers are restored from that stack. The manipulation of the SP and the saving and restoring of registers is assembler is easy: saving is pushing, restoring is popping, and the SP is accessible like any other register. We added the LR tot he saved regsiters and the PC tot he restored registers, to make the function return tot he correct location.

|  |  |
| --- | --- |
| **switch\_from\_to(**  **int \*store\_old\_sp,**  **int next\_sp**  **);**  **save registers on the (old) stack**  **\*store\_old\_sp = sp;**  **sp = next\_sp;**  **restore registers from the (new) stack**  **}** | **switch\_from\_to:**  **push { r4 – r12, lr }**  **mov r2, sp**  **str r2, [ r0 ]**    **mov sp, r1**  **pop { r4 – r12, pc }** |
| **A context-switching function in (pseudo) C and in assembler** | |

The real assembler code for a Cortex M0 is a bit more complex because the push and pop instructions can’t access the high registers.

|  |
| --- |
| **switch\_from\_to:**  **// save current context on the stack**  **push { r4 - r7, lr }**  **mov r2, r8**  **mov r3, r9**  **mov r4, r10**  **mov r5, r11**  **mov r6, r12**  **push { r2 - r6 }**    **// \*store\_old\_sp = SP**  **mov r2, sp**  **str r2, [ r0 ]**    **// SP = next\_sp**  **mov sp, r1**    **// restore the new context from the stack**  **pop { r2 - r6 }**  **mov r12, r6**  **mov r11, r5**  **mov r10, r4**  **mov r9, r3**  **mov r8, r2**  **pop { r4 - r7, pc }** |
| **A context switch function in Cortex M0 assembler** |

The rtos library implements a simple coroutine class. To create a coroutine, you must specify its stack size, and the function that is its main. After that you can call coroutine\_object.resume() to start (or resume) execution of the main of that coroutine. You could use coroutines directly, but it in most cases you’ll want to use a higher level abstraction.

# Rtos



|  |
| --- |
| Subjects |
| * rtos library * create tasks |

## Create tasks

Coroutines provide the mechanism to switch from one context to another, but little more than that: when a couroutine must wait, the user must decide which other coroutine the CPU must resume. A tasking library takes care of this adminstration of coroutines (which are now called task, or sometimes threads).

To use the rtios, the user creates a number of tasks. When a task waits, the rtos marks that task as waiting and notes for how long it must wait. Then it checks its list of tasks for one that has expired its waiting time, and resumes that task. A task that has exipred its waiting time is called ready. A task can have a priority, which can be used to select the highest priority ready task when more than one task is ready. Privately, a task object contains a coroutine.

|  |
| --- |
| **template< int stack\_size >**  **class task {**  **private:**  **coroutine< stack\_size > cor;**  **. . .**  **public:**  **virtual void main() = 0;**  **. . .**  **};** |

A task inherits from rtos::thread and implements a main(). The rtos library creates a list of all tasks. When the rtos is started, it resumes the first task, and when a task waits it resumes another (ready) task. When no task is ready the rtos just keeps on looking for one.

With the rtos, a LED blinker class has a main() function that contains a never-ending for loop. For debugging purposes each task has a name, which is passed to the task constructor. In this case more than one object of the blinker task will be made, so the tasks name is a parameter of the blinker constructor.

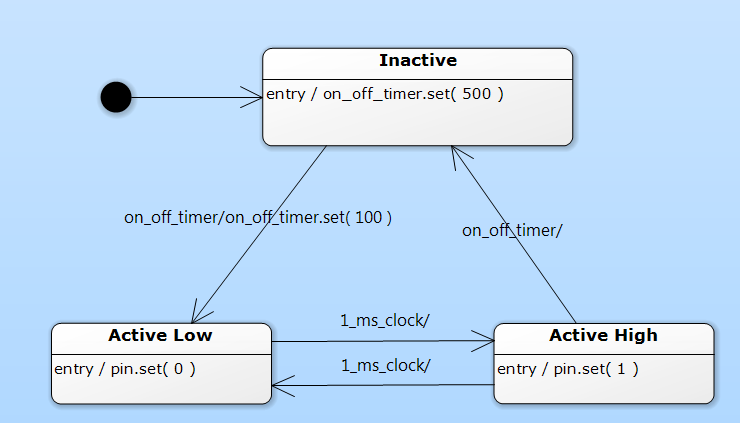
|  |
| --- |
| **class blinker : public task<> {**  **private:**  **hwlib::pin\_out & led;**  **long long d;**  **public:**  **blinker( hwlib::pin\_out & led, long long d, const char \* name ):**  **task( name ), led( led ), d( d )**  **{}**    **void main() override {**  **for(;;){**  **led.set( 1 );**  **wait\_us( d );**  **led.set( 0 );**  **wait\_us( d );**  **}**  **}**  **};** |
| **An rtos blinker task** |

The two LED blinking application instantiate two objects of the blinker class, and starts the rtos. The lines that start with (void) prevent the compiler from complaining about the blinker objects being (apparently) unused.

|  |
| --- |
| **auto led1 = target::pin\_out( target::pins::d42 );**  **auto b1 = blinker( led1, 50, "blink led1" );**  **(void) b1;**    **auto led2 = target::pin\_out( target::pins::d44 );**  **auto b2 = blinker( led2, 165,"blink led2" );**  **(void) b2;**    **rtos::run();** |
| **Create tasks and start the rtos** |

## Clocks and timers, and waiting for multiple evenst

The previous blinker class used hwlib::wait\_us(). This is fine when a task must wait for one time period, but most tasks must wait for more than one thing. Take for instance a beeper task that must make an intermittent beeping sound. There are two time periods that are relevant: the very short period between two phases of the beep signal, and the longer periods for which the beeping signal is active and inactive. This is expressed in the following STD (State Transition Diagram).



The beeping uses a clock, which is a timing element that ‘fires’ at a fixed rate, which is set when the clock is constructed. The active/inactive alternation uses a timer, which is a timing element that is set to ‘fire’ once after the specified amount of time. A timer is more flexible (you can specify a different timeout time each time you start a timer), but also a bit more work (you must explicitly start the timer, a clock runs forever).

The clock and timer objects are private objects within the task class. They must be initialized in the initialization list of the task. For all such objects you must provide the task itself (via its ‘this’ pointer), and an (optional) name (used for debugging). For a clock, you must also specify its interval, in µs (all times in the rtos are in µs).

|  |
| --- |
| **class beeper : public rtos::task<> {**  **private:**  **rtos::clock one\_ms\_clock;**  **rtos::timer on\_off\_timer;**  **...**    **public:**  **beeper:**  **task( "beeper" ),**  **one\_ms\_clock( this, 1'000, "one ms clock" ),**  **on\_off\_timer( this, "on off timer" )**  **{}**    **...**  **};** |
| **Using a clock and a timer** |

A classic way to translate a state diagram to a task is to make an enumerate for the states, and have a big switch statement in the for(;;) loop that selects on this enumerate. In each such case, you code the entry actions of the state, wait for the event that leaves the state, perform the actions that are associated with leaving the state and with the state change, and finally set the new state. Don’t forget the final break stement!

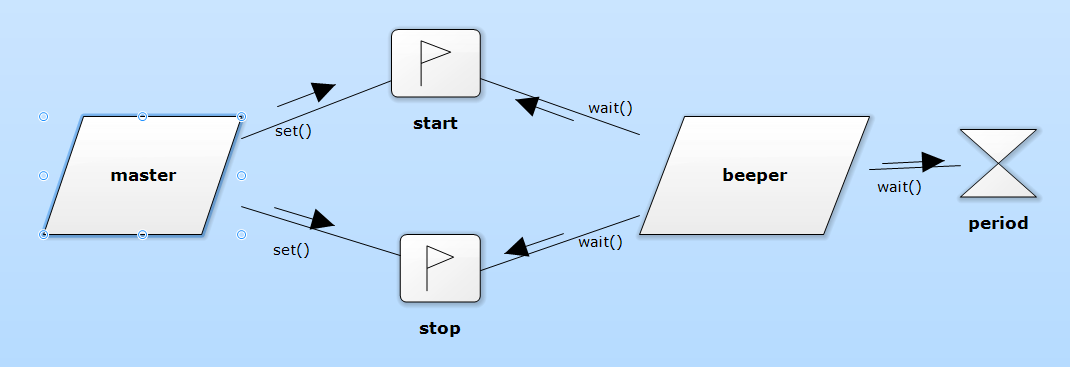
|  |
| --- |
| **class beeper : public rtos::task<> {**  **private:**  **enum class states { inactive, active\_high, active\_low };**  **states state;**  **. . .**  **public:**  **. . .**  **void main() override {**  **state = states::inactive;**  **for(;;){**  **switch( state ){**    **case states::inactive:**  **on\_off\_timer.set( 500'000 );**  **wait( on\_off\_timer );**  **on\_off\_timer.set( 100'000 );**  **state = states::active\_high;**  **break;**    **case states::active\_high:**  **lsp.set( 1 );**  **wait( one\_ms\_clock );**  **state = states::active\_low;**  **break;**    **case states::active\_low:**  **. . .**  **}**  **}**  **}**  **};** |
| **A task that implements an STD** |

This works fine for all states except active\_low, because there are two events out of that state: the timer and the clock. To wait for more than one event the rtos wait() function must be used, which can wait for an addition of waitables. A clock and a timer are both waitables. The wait() function returns the waitable that ‘happened’, and in the case of a clock or timer also ‘consumes’ that event. (If both are expired, one will be consumed and returned, and the other will not be consumed). You can use the result of the wait call to decide what to do.

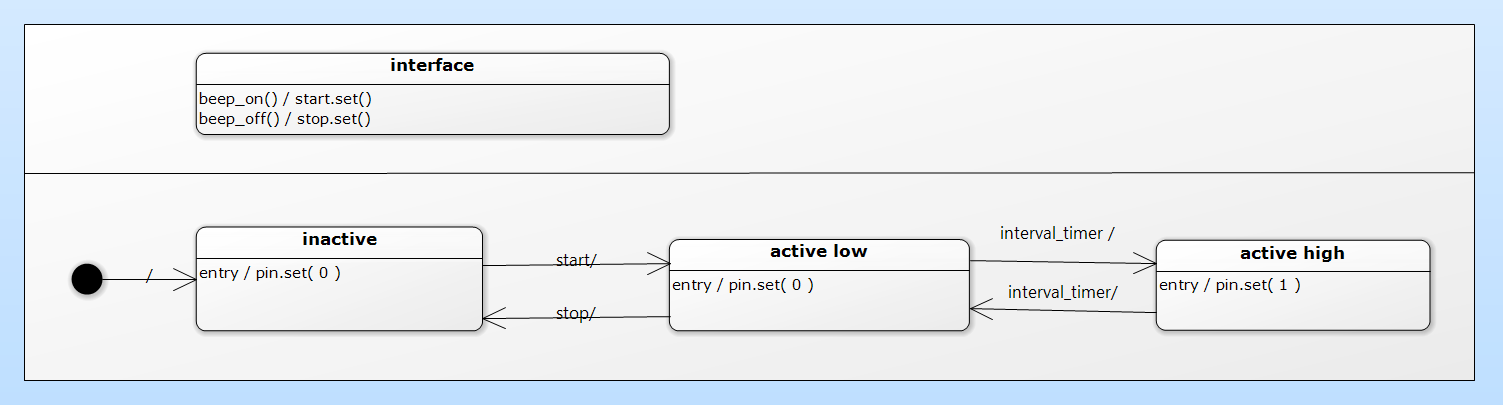
|  |
| --- |
| **case states::active\_low:**  **lsp.set( 0 );**  **auto event = wait( one\_ms\_clock + on\_off\_timer );**  **if( event == on\_off\_timer ){**  **state = states::inactive;**  **} else {**  **state = states::active\_high;**  **}**  **break;** |
| **A state that must wait for and handle more than one event** |

## Event flags and external interfaces

The pervious beeper combined the generation of the sound and the on-off switching in one task. It might make sense to separate these two concerns: one task beeps, and it offers methods to start and stop the beeping. A second task switches the beeping on and off by calling these methods. Note that these methods are part of the beeper class, but are not run by the beeper task. Interaction between two task is never direct, but always through a synchronization element. One such element is the event flag. The owner of an event flag can wait for the flag, and other tasks can set the flag. When a task waits for a flag, another task can set it, the waiting task continues, and the flag is (automatically) reset.



The concurrency diagram shows the methods that are called on the synchronization objects. A state diagram of a task that has an external interface distinguishes between what is done by the task itself (lower half), and what is done by other tasks (upper half). The synchronization objects (in this case: the two event flags) are the connection between these two halves. The part after the / in the upper half contains operations that ‘set’ these synchronization objects, the lower half contains operations that wait on these objects.



The ‘events’ in the upper half (the external interface) are the names of the functions that can be called by other tasks, and the actions that are performed in response to these events are the bodies of these functions.

|  |
| --- |
| **class beeper : public rtos::task<> {**  **private:**  **hwlib::pin\_out & lsp;**  **rtos::clock one\_ms\_clock;**  **rtos::flag start, stop;**  **enum class states { inactive, active\_high, active\_low };**  **states state;**    **public:**  **beeper( hwlib::pin\_out & lsp ):**  **task( "beeper" ),**  **lsp( lsp ),**  **one\_ms\_clock( this, 1'000, "one ms clock" ),**  **start( this, "start" ),**  **stop( this, "stop" )**  **{}**    **void beep\_on(){ start.set(); }**  **void beep\_off(){ stop.set(); }**    **void main() override {**  **state = states::inactive;**  **for(;;){**  **switch( state ){**    **case states::inactive:**  **wait( start );**  **state = states::active\_high;**  **break;**    **case states::active\_high:**  **lsp.set( 1 );**  **wait( one\_ms\_clock );**  **state = states::active\_low;**  **break;**    **case states::active\_low:**  **lsp.set( 0 );**  **auto event = wait( one\_ms\_clock + stop );**  **if( event == stop ){**  **state = states::inactive;**  **} else {**  **state = states::active\_high;**  **}**  **break;**  **}**  **}**  **}**  **};** |
| **A beeper task that can be started and stopped by another task** |

## Pools and mutexes

In the simpe rtos that we use a pool is nothing more than a variable that is shared between tasks. Its main purpose is to make such sharing explicit. This serves the reader of the design and the code, and would make it easier to modify the code to run on an rtos or hardware platform where more actions are needed to share data between tasks.

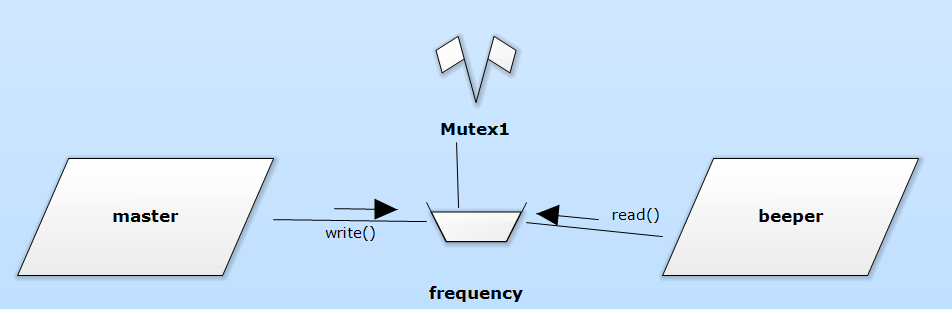
A pool is a class template. The template parameter is the type of the data that is stored in the pool. This can be a built-in type like an int, but it can also be a user-defined type like a struct or even a class. The pool operations are read() and write(), which read and write the entire data object.

The rtos we use is cooperative (non-preemptive), which means that switching between tasks can happen only when an rtos call (including hwlib::wait\_\*) is done. When a preemptive rtos is used, task switching can happen at any moment, including halfway a C++ statement. This can have interesting consequences when a task writes to a shared piece of memory, and halfway this action the rtos switches to another task, which then reads the same piece of memory. The memory as seen by the second task can be halfway updated.

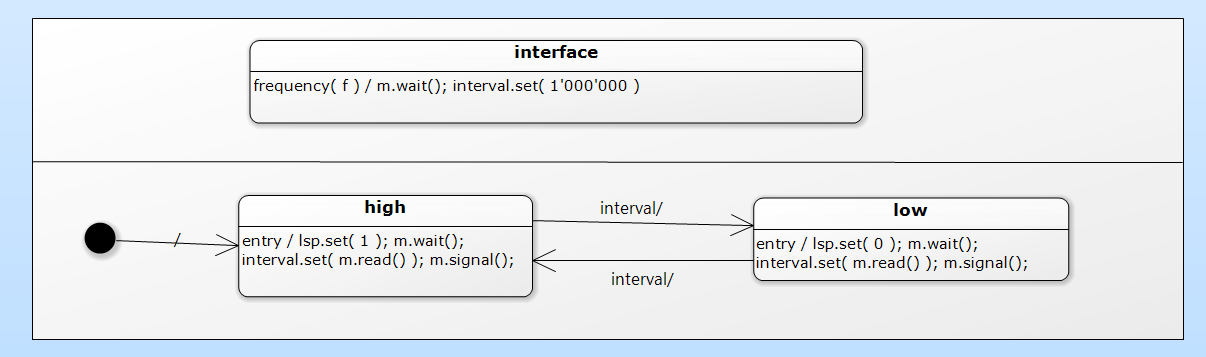
As an example, if the writing task overwrites a message “Hello world” with a different text, for instance “Keep smiling”, the reading process could read “Keepo world”. In most cases this is highly undesirable. A very crude solution is to stop preemptive task switching for the the duration of such a ‘critical’ read or write, but this would also impact other tasks (that have nothing to do with this piece of shared data). A better solution is to make sure that only one task has access to the shared data. This can be achieved by a mutex (Mutual Exclusion flag). A task that needs access to the shared data claims the flag, access the data, and releases the flag.

### Pool and mutex example

Imagine that a task must be able to change the beep frequency of the beeper task. For this, the beeper can offer a frequency() function. This function calculates the delay period, and writes it to a pool. The beeper task reads the pool each time it starts its period timer. In the concurrency diagram the mutex is shown ‘attached’ tot he pool. This attachment implies that all access to the pool must claim (and free) the mutex.



Like event flags, pools are used between the interface functions of a task class and the main() of the task. Each access (read or write) of the pool is protected by a mutex to prevent simulatenous reading and writing. Both the pool and the mutex are hidden inside the beeper task.



A pool and a mutex are not waitables, hence you can’t use them in a wait() call. In a state diagram you won’t find them as events, and in their constructors you don’t pass the this pointer of the task.

|  |
| --- |
| **class beeper : public rtos::task<> {**  **private:**  **hwlib::pin\_out & lsp;**  **rtos::pool< int > interval\_pool;**  **rtos::mutex interval\_mutex;**    **public:**  **beeper( hwlib::pin\_out & lsp ):**  **task( "beeper" ),**  **lsp( lsp ),**  **interval\_pool( "interval pool" ),**  **interval\_mutex( "interval mutex" )**  **{}**    **void frequency( int n ){**  **interval\_mutex.wait();**  **interval\_pool.write( 2'000'000 / n );**  **interval\_mutex.signal();**  **}**    **void main() override {**  **for(;;){**  **lsp.set( 0 );**    **interval\_mutex.wait();**  **auto w = interval\_pool.read();**  **interval\_mutex.signal();**    **hwlib::wait\_us( w );**    **lsp.set( 1 );**    **interval\_mutex.wait();**  **w = interval\_pool.read();**  **interval\_mutex.signal();**    **hwlib::wait\_us( w );**  **}**  **}**  **};** |
| **A beeper that can have its frequency changed** |

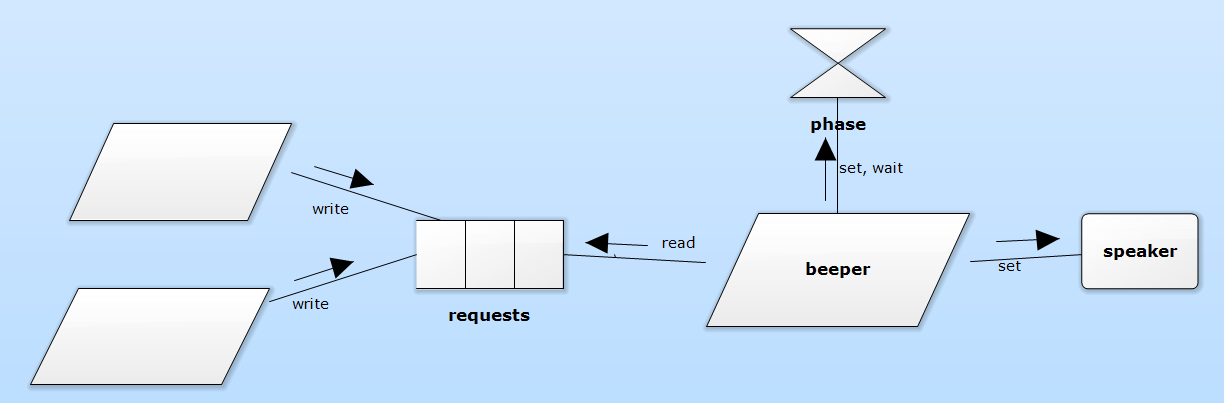
## Channels

A channel is a queue of data elements. The task that owns the channel can read a data element from the head, other tasks can write data elements to the tail. Reading from a channel is blocking: when the channel is empty (contains no elements) the reading task will be blocked until an element is available. Reading an element consumes that element: it is removed from the channel. If you don’t want to be blocked by reading from a channel you can include the channel in a wait() call.

When the wait() call returns the channel, it contains at least one data element and hence a read() on it won’t block.

### Channel example

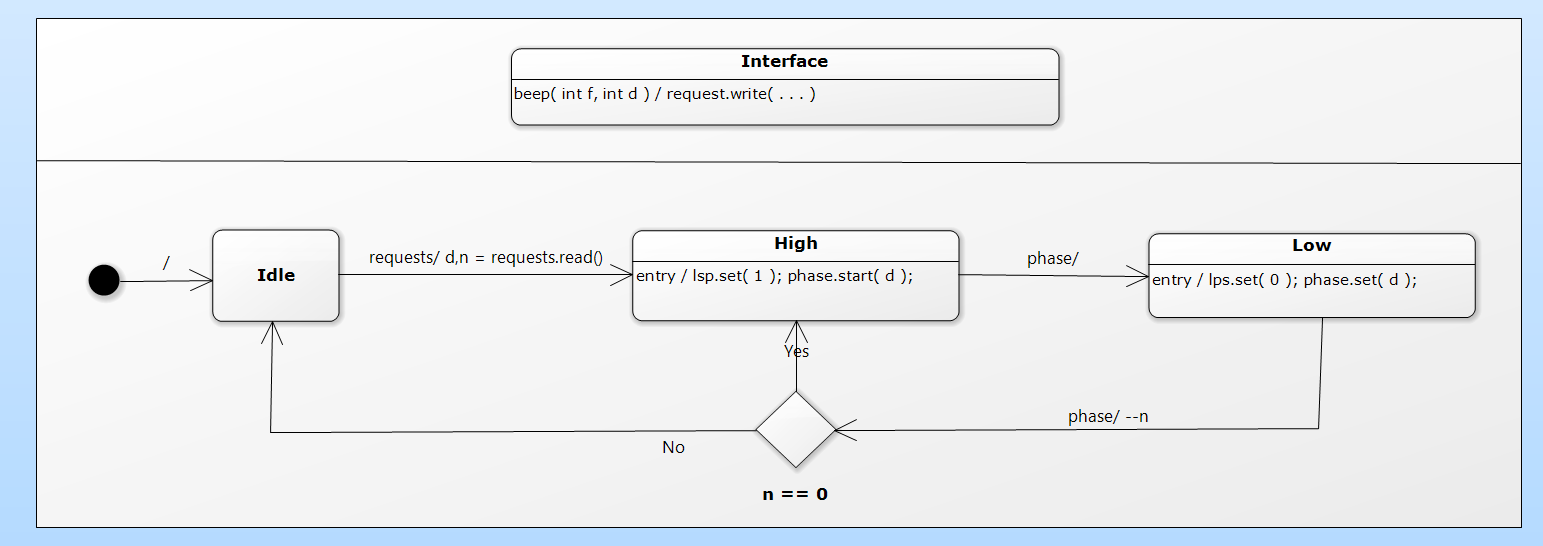
The following beeper task accepts requests to beep. These requests are queued in a channel, because the requesting task must probably do other things while the beeping is done (or is waiting for its turn). The beeper task obviously needs a timer to time the beeping.



Internally, this request is translated to a stuct with two elements: the number of cycles and the duration (time to wait) of a half-cycle. This struct is put into the requests channel.

|  |
| --- |
| **struct request{ int p; int n; };**  **rtos::channel< request, 10 > requests;**  **void beep( int t, int f ){**  **request req;**  **req.p = 2'000'000 / f;**  **req.n = t / ( 2 \* req.p );**  **requests.write( req );**  **}** |
| **A request channel and the external interface that hides it** |

The details of this translation are omitted in the next STD, but in a complete design they should be specified somewhere (either in the STD itself, or in the accompanying description).



A channel has a maximum number of elements. When the channel is full, extra writes to it will be ignored. A channel is a class template. The template parameters are the type of the data element, and the number of data elements that can be stored.

The task itself simply reads from this channel, and does the beeping as instructed by the data read from the channel. When the task waits for a new beep, it waits only on the channel, so there is no need to do a wait: a read on the channel will do the blocking and reading in one go.

The concurrency diagram and the STD showed a timer and set and wait operations on that timer. In this example, this timer is always used in a straightforward way: it is set, and then the state waits on just that timer. In such a case the timer can be omitted and a wait call can be used.

|  |
| --- |
| **class beeper : public rtos::task<> {**  **private:**  **hwlib::pin\_out & lsp;**  **struct request{ int p; int n; };**  **rtos::channel< request, 10 > requests;**    **public:**  **beeper( hwlib::pin\_out & lsp ):**  **task( "beeper" ),**  **lsp( lsp ),**  **requests( this, "requests" )**  **{ }**    **void beep( int t, int f ){ ... }**    **void main() override {**  **for(;;){**  **// could do: wait( requests );**  **auto req = requests.read();**  **while( req.n-- > 0 ){**  **hwlib::wait\_us( req.p );**  **lsp.set( 1 );**  **hwlib::wait\_us( req.p );**  **lsp.set( 0 );**  **}**  **}**  **}**  **};** |

# The NEC IR protocol

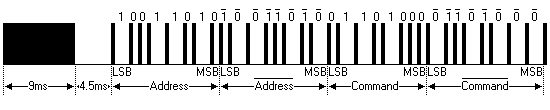
|  |
| --- |
| Subjects |
| * The NEC IR protocol * A design for an application that receives and handles NEC IR commands |

## Protocol descriptipon

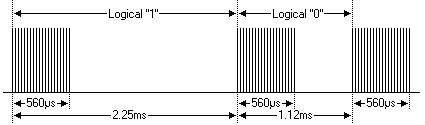
The NEC IR (Infra RED) protocol is a format for IR data transmission from a remote control unit to a receiver (for instance a TV set). The following description is based on <http://www.sbprojects.com/knowledge/ir/nec.php>. This site contains descriptions of a number of IR protocols.

The protocol uses IR modulated (switched on and off) at 38 kHz. The commonly used reveivers filter this modulation out, but when you create an IR signal you must apply this modulation: an IR LED won’t do this for you.

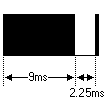
A message starts with a 9 ms signal and a 4.5 ms pause, followed by 4 \* 8 data bits. Byte 1 is the address (the code of the device for which the message is intended), byte 3 is the command to the device. For error detection, byte 2 is the bitwise inverse of byte 1 and byte 4 is the inverse of byte 3.



Each bit is transmitted as a pulse of 560 µs, followed by a pause of 1690 µs for a 1, or 560 µs for a 0. Note that the message ends with one more pulse, otherwise you couldn’t measure the pause of the last bit.

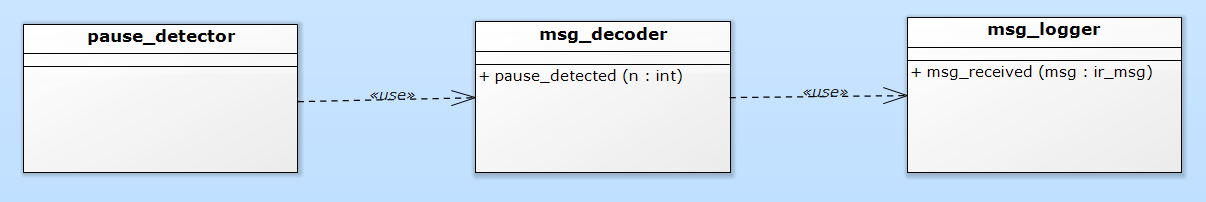


When you press a button on the control unit, it will transmit the described message. If you keep pressing the button, the control unit will (after the first message) send ‘repeat’ messages that indicate that the button is still pressed. This repeat message is the same for all buttons. It consist of a 9 ms signal, a 2.25 ms pause, and a 560 µs pulse.



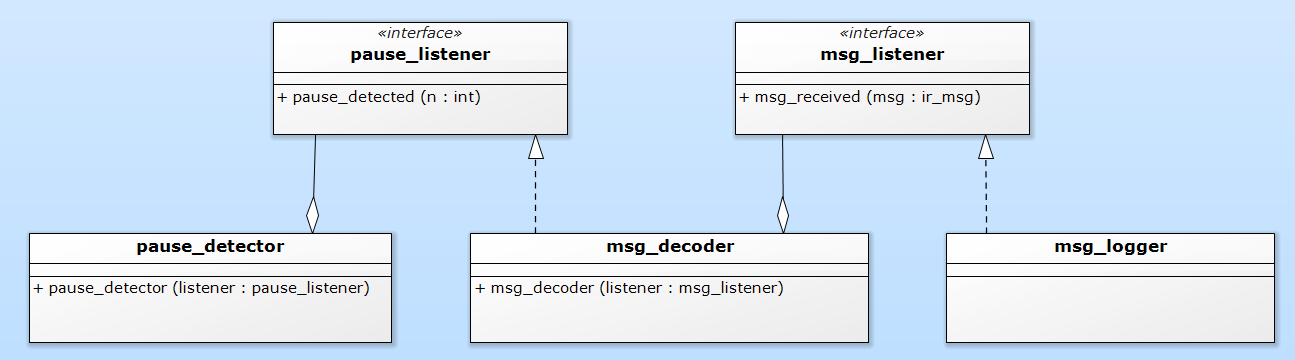
## A design of an IR receiver application

The following initial class diagram shows decoder-and-logger for NEC IR messages. The pause\_detector detects the length of pauses in the IR signal. (Note that a message can be fully decoded from only these pauses!) The msg\_decoder decodes a train of such pauses into messages. When a message is found to be correct (the inverted values are indeed the inverse of the address and command values) the message is passed onto the logger, which displays the messages on the OLED display. In a real application, the logger would be the object that handles IR commands.

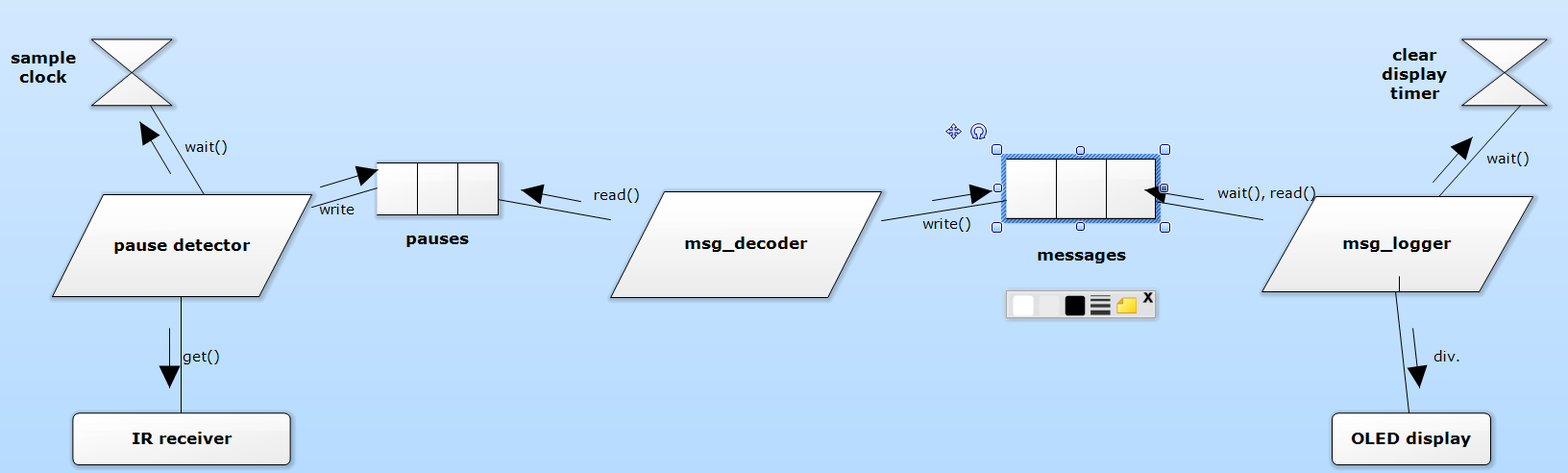


In this initial class diagram the calling (using) is bottom-up: lower-level classes call higher-level classes. This isn’t a good situation: a class can depend on lower-level classes, but should not depend on higher-level classes.

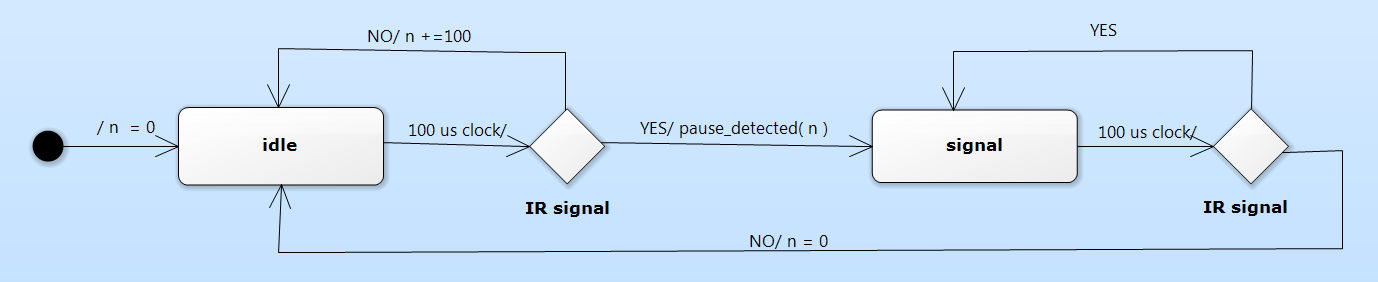
To avoid this inverse coupling the listener pattern is used (twice): the relevant interface part of the higher level class is put in an interface class. The lower level class depends on this interface, the higher level class implements it. At run-time, the lower level objkect is informed which higher level object it must use. In this case this is done by the constructor call.



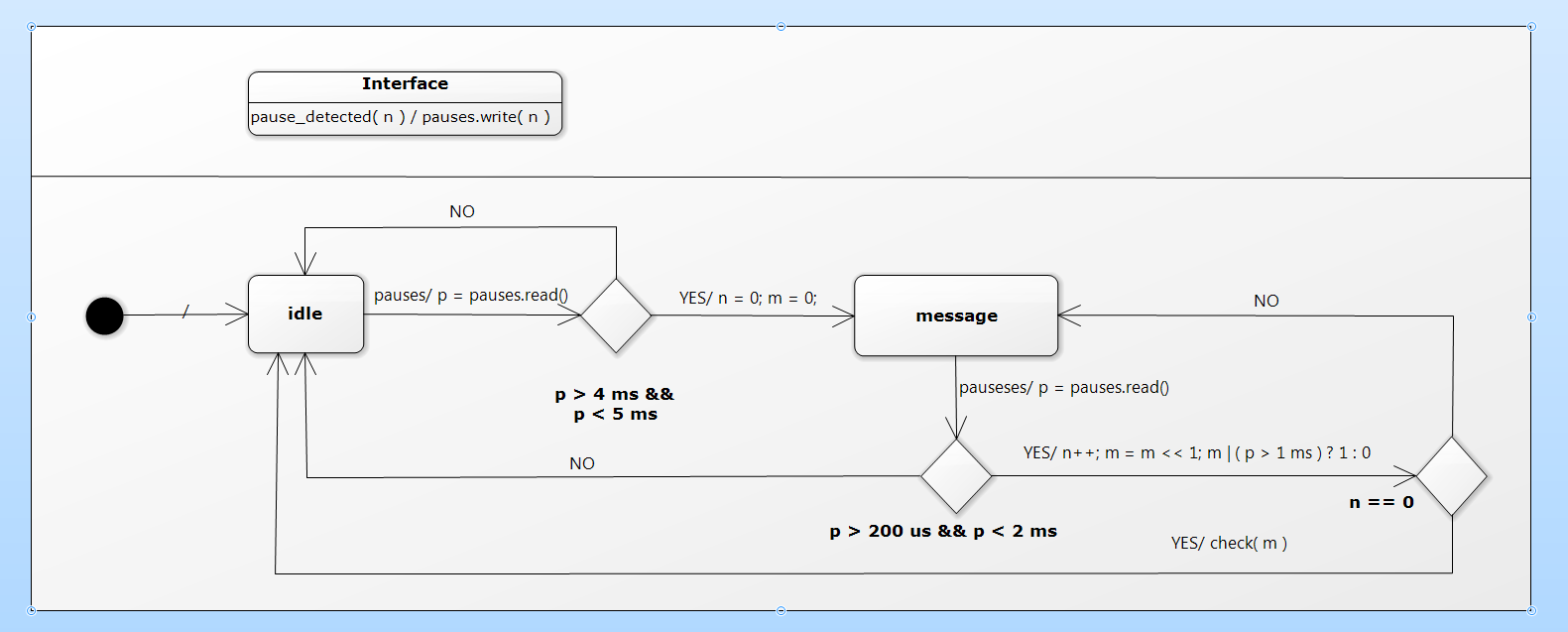
The concurrency diagram shows the three active objects as tasks. Let’s assume that the pauses can be detected faster than they can be decoded into messages, and that valid messages can arrive faster than they can be displayed. In that case is makes sense to use buffering channels) between the tasks.



The pause\_detector polls the IR receiver at fixed intervals, and reports the length of the pauses (accurate to the 100 us sampling interval) to the msg\_decoder. A clock is used to time the sampling. The STD has only two states: a pause is in progress, or an IR signal is in progress. While a pause is in progress, its length is measured, and when it ends it is handed over to the next task. When an active IR signal is being detected, nothing has to be done, because only the length of the pauses is needed by the next task.



The msg\_decoder receives the pause lengths. Its interface puts the lengths (just integers) into a channel. The STDs of the task itself has two states: idle, and message in progress. From the idle state, only an appropriate pause can cause a transition to the message state, all other pauses are ignored. In the message state, 32 valid pauses are accepted and stored as bits. An invalid (tto short or too long) pause will cause a transition back to the idle state, dropping a partially received message. After the 16’th bit, the message is processed. This amounts to checking the address and command against their inverted values. If OK, the message is passed on to the next taks, otherwise it is dropped. This processing has no real-time aspect, hence it is not shown in detail in the STD, which mentions it only as check( m ).



The final task displays the valid received messages on the OLED. It shows a message until a next message is received, or for 5 seconds when no new message is received.

