Wireless Relay Module

Model H0738

Programming Guide & Reference

version 1.0 — March 2024

CompuPhase

1e Industriestraat 19, 1401VL Bussum, The Netherlands
telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com
WWW: https://www.compuphase.com

Typeset with TEX in the "Adobe Source" typeface family.

Contents

Overview

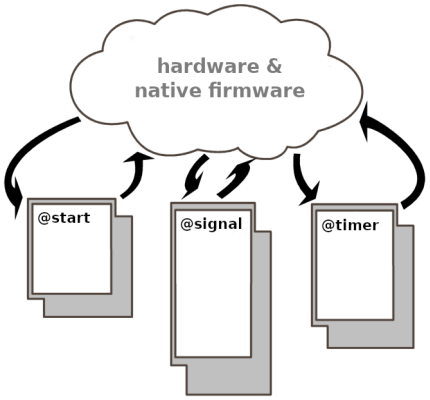
The wireless relay module, model H0738, has built-in functionality —as described in its user guide. Out of the box, when you press a button that is linked to the module, the relay is actuated for a short duration ("actuated" meaning that the relay closes). The duration that the relay stays actuated, is set with a rotary switch on the module.

You can override the built-in functionality by creating a script. The script responds to events, such as button presses, ticks of a timer that you can set up, or data received on the USB port. The relay module uses the "PAWN" scripting system.

PAWN is a general-purpose scripting language. This reference is not a guide on the language itself —please see the companion document "The PAWN booklet — The Language" for a tutorial.

Event-driven programming

The relay module follows an "event-driven" programming model. In this model, your script does not poll for events, but instead an event "fires" a function in your script. This function then runs to completion and then returns. In absence of events, the script is idle: no function runs.



For example, when the module receives a signal from a wireless button, this fires the function "`@signal`" in the script —provided that the script has this function. This function then runs, and can perform operations like actuating or releasing the relay or setting a timer. After the `@signal` function is done, it simply returns or exits the script. The system is now idle, but another event may wake it up. The event-driven programming model thereby creates reactive and/or interactive programs. The general manual "The PAWN booklet — The Language" has more details on the event-driven model.

The following script is a first example of a script for the relay module. A press on a wireless button actuates the relay for 15 seconds; an overly simplistic script, perhaps, but it already demonstrates an extension over the capabilities of the built-in functionality: the rotary switch only allows durations of up to 9 seconds.

```
@signal(button)
    {
    setrelay 1      /* actuate the relay */
    sleep 15000     /* delay is in millisconds */
    setrelay 0      /* release the relay */
    }
```

When a function in the script is running, no other event can be handled. That is, while the script is busy inside the `@signal` function, a new press may be queued, but it is not executed. Only after the function has returned, will any pending event be handled. Functions do *not* interrupt or *pre-empt* each other.

Which is also to say that, if you need to respond to other events during the fifteen-second "sleep" of the above script, the "sleep" should instead be implemented as a timer event.

```
var n_countdown

@signal(button)
    {
    setrelay 1          /* actuate the relay */
    n_countdown = 15
    }

@timer()
    {
    if (n_countdown > 0)
        {
        n_countdown = n_countdown - 1
        if (n_countdown == 0)
            setrelay 0  /* release the relay */
        }
    }

@start()
    {
    settimer 1000       /* interval is in milliseconds */
    }
```

In this second example, the `@signal` actuates the relay and sets a counter variable to 15 —then it returns, without waiting. Instead, the `@timer` event runs every second, and this function decrements the countdown variable until it reaches zero, and then releases the relay.

A power-up of the relay module causes a `@start` event. This function therefore runs only once, but it runs before any other function. Here, it is used to configure the interval of the timer event.

USB interface

The wireless relay module is powered through the USB connector. For transferring a script into the module, the module must be connected to a PC or workstation with a USB cable.

After a script is stored on the module, the module does *not* need a USB connection to run. It only requires power, so you can use a mains power supply or a power bank. The script remains stored (in Flash ROM) on the module when power is removed.

The exception to the rule "the module does not need a USB connection to run a script" is when the script receives commands through the USB port, or transmits messages back to the host. The module *can* be used as a USB-controlled relay.

The USB port emulates a serial port. Modern operating systems (e.g. Microsoft Windows as of Windows 10) have built-in support for virtual serial ports. Hence, no drivers need to be installed on the PC or workstation. For earlier versions of Microsoft Windows, the operating system also provides a driver for virtual serial ports, but you need to run an install in `INF` file to "link" the USB device to the driver. A suitable `INF` file is provided in the "setup" directory below where the software kit for the relay module is installed.

When opening a virtual serial port in a serial terminal or other application, you will typically need to specify a baud rate and the number of data & stop bits. These are legacy settings, and they are irrelevant in the case of the relay module —the module will accept a connection at *any* baud rate and frame configuration, because it *always* uses the USB protocol & speed.

I/O pins

The relay module has two general-purpose digital input/output pins. These are on a terminal connector with three connections; the third connection is for the "ground" signal.

The built-in functionality of the module does not use these I/O pins in any way. A user script, however, can read these pins, and/or set them.

The I/O pins have an internal pull-up. When used as inputs, a potential-free contact (such as a standard switch) can be directly connected between the I/O pin and the ground.

The pins use 3.3 V TTL logic levels; when used as input, they are 5 V tolerant.

Other tools and documents

This guide focuses on two software tools for scripting the relay module, and it describes the hooks that the hardware provides for scriptable functionality. There

are a few additional software utilities and resources in the development kit, which may be useful on their own.

The utility "H0738Config" allows you to check and configure the relay module. When the relay module is connected to a PC/workstation with a USB cable, the module presents itself as a virtual serial port (see also section USB interface, above). However, *which* serial port (or COM port) gets assigned to the module, depends on the operating system and on what other serial devices are (or have been) attached to the PC or workstation. The configuration utility scans the system and locates the appropriate port.

The utility also allows you to link and unlink wireless buttons, as well as placing the button links in a particular order. It is not necessary to have the wireless buttons at hand, when using this procedure, but you do need the serial numbers and pin codes for the buttons.

For general information on the relay module, a quickstart guide is available in the "doc" subdirectory below the path where the development kit is instaled. This quickstart guide contains the specifications of the module, and has information on its use. Note that the quickstart guide focuses on the built-in functionality of the relay module (i.e. when no script is active).

The PAWN language is exhaustively documented in the document "The PAWN booklet — The Language". The tutorial section is not immediately applicable the the relay module (because the tutorial assumes a PC or device with a display), but the language syntax and semantics are fully described.

Creating scripts

As a general note: when a script is uploaded into the relay module, the script takes over the built-in functionality completely. That is, when you create a script that responds to I/O pins, but omits a function (or "event block") to respond to a signal of a wireless button, the module will no longer respond to button presses *at all*, regardless of whether the button is linked to the interface.

This guide is mostly aimed at the core PAWN scripting support built into the relay module. There is, however, a higher-level, graphical programming tool for programming the relay module as well: Pawn Blocks Designer. The development kit contains both the visual Designer and the source-code based PAWN IDE.

Scripts made in Pawn Blocks Designer can also be opened in the PAWN IDE, but the inverse is not true.

Programming examples

Several programming examples are provided together with the development kit. You will find these in the "examples" subdirectory of where the kit is installed.

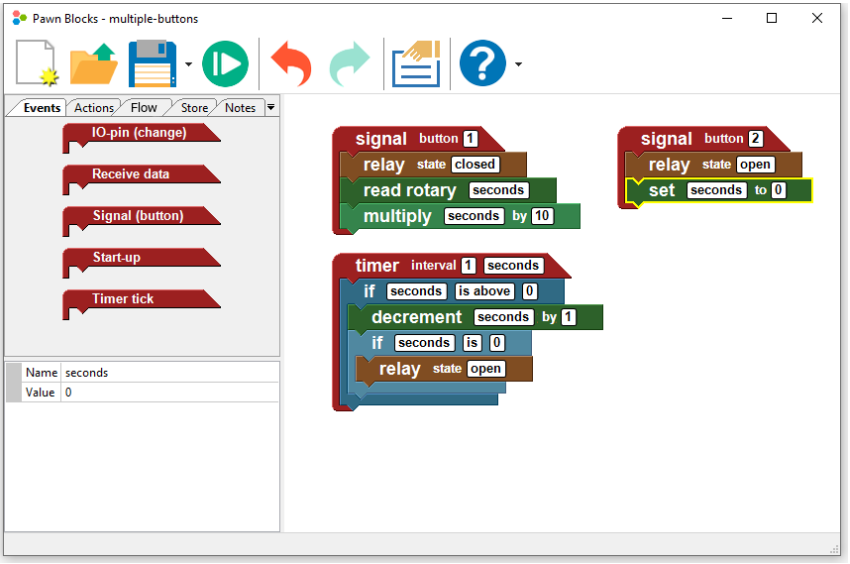
The examples are suitable for both Pawn Blocks Designer and the PAWN IDE.

Pawn Blocks Designer

With Pawn Blocks Designer, you create scripts by *dragging* blocks from the "tool panel" (left section of the application window), and *dropping* them onto the "worksheet". By grouping blocks together and setting parameters for each block, you structure the script.

Pawn Blocks Designer avoids common programming errors, because it only allows you to attach and group blocks in a way that is syntactically correct. While you drag a block, the designer visually indicates where you may drop it. In the same vein, the designer catches some parameter errors early on. If you are new to programming, this graphical programming method helps you get a smooth start.

The blocks are divided in four categories in the tool palette: Events, Actions, Flow and Store. The "event" blocks reflect the events that the firmware of the device may invoke in the script. For the relay module, we will find the "Signal", "Start-up" and "Timer tick" blocks, mirroring the `@signal`, `@start` and `@timer` event functions used in the example on page 2.



The Actions TAB has blocks that change a setting or perform an activity in the the device: actuating or releasing a relay, toggling an I/O pin low or high, turning a LED on or off, . . .

The Flow TAB has only a few blocks, for program flow. Here are the if. . . else blocks for conditional execution as well as a loop block.

In the Store category are blocks that set or change variables. These variables can be set from literal values, or be queried from the device. If you want to check on the status of an I/O pin, for example, you will find the appropriate block under this TAB.

Blocks with (nearly) matching names may appear in multiple categories, depending on how it is used. For example, to set an I/O pin, you will find the appropriate block in the Action category, but to query the state of an I/O pin, you will need a block from the Store TAB. Both blocks are called "io-pin", but are distinguished by colour.

There is a fifth TAB in the tool palette: Notes. This TAB does not contain tools, but a general-purpose edit field. It allows you to enter notes about the script: what it does, how it functions, and any other details. The notes are stored as "comments" in the script file, and they do not affect the functioning (or size) of the script in any way.

The Toolbar

The toolbar of the Pawn Blocks Designer has the standard tools for New, Open and Save, but next to that is a tool to transfer the script to the hardware device. See the

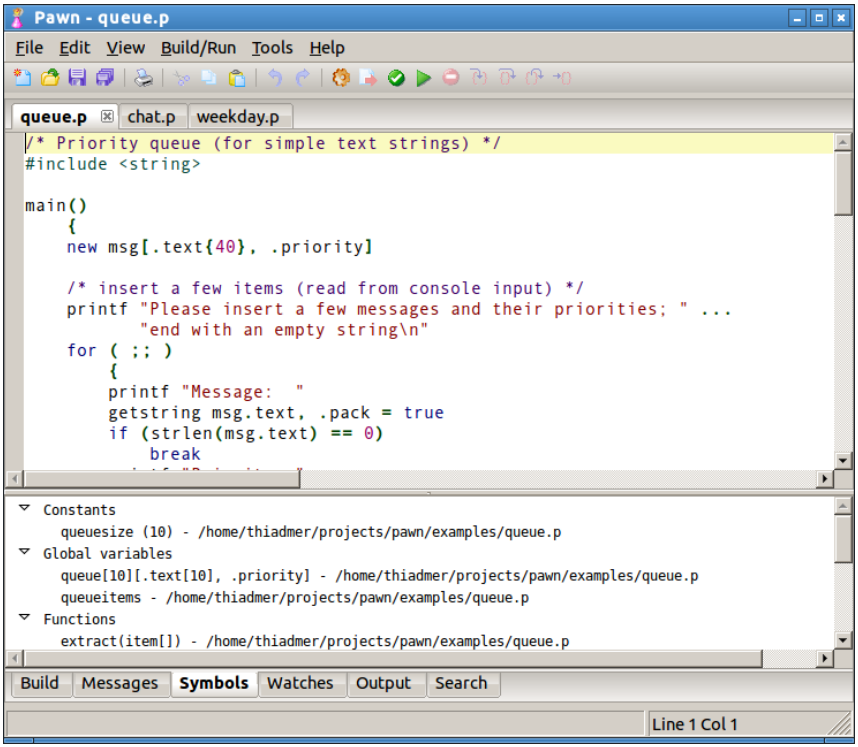
section Transferring scripts over USB for more information on transferring the script to the relay module. Note that you do not need to have the relay module connected to the PC/workstation for creating the script, but to transfer it, the relay module needs to be connected.

Wedged between the Undo/Redo buttons and the Help button, is a tool for configuring the relay module. This configuration tool provides an alternative way to link or unlink buttons, or to clear the relay module.

Pawn Blocks Designer saves the script as PAWN source code (with attributes in comments). You can therefore open, modify and rebuild the script in the PAWN IDE. However, you may not be able to re-open the script in Pawn Blocks Designer —in any case, the changes made in the PAWN IDE will be lost when you do re-open the script in Pawn Blocks Designer.

Pawn IDE

You can use any text editor to write PAWN scripts, and then compile these and transfer them to the device. However, the PAWN IDE provides syntax highlighting and code completion specifically for PAWN, and it also integrates with the other PAWN tools. Thus, we recommend that you use the PAWN IDE for creating scripts.



The IDE is for general-purpose programming. To generate the appropriate scripts for a specific device, the IDE uses a "Target host" configuration. These can be set in the options; see the menu Tools / Options. For the relay module, the target host must be set to "H0738". You can also check the caption of the main window of the PAWN IDE: it should say "Pawn[H0738]" (followed by the name of the active source file).

As a programming system, PAWN consists of the "language" and a "library". The language is standardized and common for all applications. The library gives access to all the functionality that the host application/device provides. That being the case, the library is typically highly specific to the system into which PAWN is embedded. If you pick up a different device that supports PAWN scripting, you will likely need to get acquainted with the *library* of that other device, but the *language* stays the same.

This library is, in turn, split into "public" functions and "native" functions. Public functions are functions that you implement in the script, but that are called from the firmware of the device. Public functions are the "event" functions that the section Event-driven programming referred to. Commonly, public functions start with the character "@". The public functions that the relay module supports, are documented in the chapter Public functions.

Native functions are routines that are embedded in the firmware, but that you can call from the PAWN script. In other scripting languages, these are called "foreign functions". There is a native function, for example, to switch the relay on or off. The native functions supported by the relay module, are in the chapter Native functions.

Transferring scripts over USB

There are three ways to transfer a script to the relay module:

- With a command-line utility —but you need to make sure yourself that you only transfer a valid script to the module.
- Using the "Build & Export" tool in Pawn Blocks Designer —but which requires that the script is created in Pawn Blocks Designer as well.
- Using the "Transfer" button in the PAWN IDE.

In all cases, the first step is to connect the relay module to the PC/workstation with a USB cable. All three methods automatically locate the serial port (that the module gets assigned by the operating system). This may fail, though, when multiple modules are connected to the same PC at the same time. For programming the relay modules, it is recommended to connect a single unit to a PC at a time.

Public functions

@iopin

An IO-pin toggled state

Syntax: `@iopin(pin, status)`

`pin`	The pin number, 0 or 1.
`status`	The new status, 0 for low, 1 for high.

Returns: None, the return value of this function is ignored.

Notes: This event fires when the pin transitions fro low-to-high or high-to-low. When a pin stays high or low, no event is generated.

When a pin state is set with `setiopin`, the pin is configured as "output". Events are *not* generated for output pins.

See also: `getiopin`, `setiopin`

@receive

Received data from the USB port

Syntax: `@receive(const data, length)`

`data`	The received data, as a byte array.
`length`	The number of bytes that were received (and stored in the `data` array).

Returns: None, the return value of this function is ignored.

See also: `transmit`, section USB interface

@signal

Received a signal from a wireless button

Syntax: `@signal(button)`

`button`	The index of the wireless button, whose signal is received. This is a value between 1 and 4.

Returns: None, the return value of this function is ignored.

Notes: Wireless buttons must first be linked to the relay module. Up to four buttons can be linked. The order in which the buttons are linked, is also the sequence number of the `index` parameter.

Each wireless button has a unique serial number. You can match an index to a serial number (of a linked button) with `buttonserial`.

See also: `buttonserial`

@start — Power-up or reset

Syntax: `@start()`

Returns: None, the return value of this function is ignored.

Notes: `main` is an alternative name for function `@start`.

@timer — A timer interval passed

Syntax: `@timer()`

Returns: None, the return value of this function is ignored.

Notes: This function executes after the configured delay or interval expires. See `settimer` to set the delay or interval.

If the timer was set as a "single-shot", it must be explicitly set again for a next execution for the `@timer` function. If the timer is set to be repetitive, `@timer` will continue to be called with the set interval until it is disabled with another call to `settimer`.

See also: `settimer`

Native functions

buttonserial

Return the serial number of a linked button

Syntax: `buttonserial(button)`

`button` The sequence number of the button; a value between 1 and 4.

Returns: The serial number of the button, *without* the letter prefix. When no button is linked at the specified sequence number, the return value is zero.

Notes: The function only returns the last three digits of the serial number. That is, if the linked button's serial is "`P123`", this function returns 123.

See also: `@signal`

getiopin

Read the indicated I/O pin

Syntax: `getiopin(pin)`

`pin` The pin number, either 0 or 1.

Returns: The logical level of the specified I/O pin: 0 for "low" or 1 for "high".

Notes: This function always returns the current logical level of the pin, regardless of whether the public function `@iopin` is defined.

On start-up, the I/O pins are configured as "input". If a pin was configured as "output" (by `setiopin`), calling this function re-configures it as "input".

See also: `@iopin`, `setiopin`, section I/O pins

random

Return a pseudo-random number

Syntax: `random(max)`

`max` The limit for the random number.

Returns: A pseudo-random number in the range `0...max-1`.

rotaryswitch

Read the rotary switch

Syntax: `rotaryswitch()`

Returns: A value between 0 and 9 (the setting of the rotary switch).

See also: `getiopin`

setiopin

Set the indicated I/O pin

Syntax: `setiopin(pin, status)`

`pin`	The pin number, either 0 or 1.
`status`	The new status for the pin. This is a logical value: 0 or 1.

Returns: This function always returns 0.

Notes: On calling `setiopin`, the specified pin will be configured a "output". (After reset, all pins are configured as inputs.)

Logic output levels of the I/O pins are 0 V for "low" and 3.3 V for "high".

See also: `getiopin`, `setled`, `setrelay`, section I/O pins

setled

Configure a pin for input sampling

Syntax: `setled(status)`

`status`	Either 1 ("LED on"), or 0 ("LED off"). When setting this parameter to -1, the LED follows the state of the relay: it is on when the relay is actuated and off when the relay is released.

Returns: This function always returns 0.

Notes: There are two LEDs on the module; the green LED is hard-wired to indicate "power". This function controls the red LED.

By default, the state of the LED follows the state of the relay. When using this function to switch the LED on or off, this default is overruled.

The red LED is also used to indicated the progress of the button "link" procedure (after pushing the "connect" switch on the board. This function does not change the link procedure; the red LED will still blink after pressing the "connect" button.

See also: `setiopin`, `setrelay`

setrelay

Configure a pin for input sampling

Syntax: `setrelay(status)`

`status` Either 1 (relay *actuated* —meaning that the relay is closed), or 0 (relay *released* —meaning that it is open).

Returns: This function always returns 0.

Notes: By default, the red LED lights up while the relay is actuated; this can be overruled with `setled`.

See also: `setiopin`, `setled`

settimer

Configure the event timer

Syntax: `settimer(milliseconds, bool: singleshot=false)`

`milliseconds` The number of milliseconds to wait before calling the `@timer` callback function. Of the timer is repetitive, this is the interval. When this parameter is 0 (zero), the timer is shut off.

`singleshot` If `false`, the timer is a repetitive timer; if `true` the timer is shut off after invoking the `@timer` event once.

Returns: This function always returns 0.

See also: `@timer`, `tickcount`

tickcount

Return the current tick count

Syntax: `tickcount()`

Returns: The number of milliseconds since start-up of the system. For a 32-bit cell, this count overflows after approximately 24 days of continuous operation.

Notes: This function will return the time stamp regardless of whether a timer was set up with `settimer`.

See also: `settimer`

transmit

Transmit data over the USB port

Syntax: `bool: transmit(const data, length=-1)`

`data`	The data to transmit, as a byte array.
`length`	The number of bytes to transmit, or -1 if the data is a zero-terminated (packed) string.

Returns: `true` on success, `false` on failure.

See also: `@receive`, section USB interface

version

Return the firmware version

Syntax: `version()`

Returns: The “build number” of the firmware in the relay module.

Index