IRMP - english

Aus der Mikrocontroller.net Artikelsammlung, mit Beiträgen verschiedener Autoren (siehe Versionsgeschichte)

By **Frank M. (ukw)**

This is the English translation of the German IRMP documentation.

Project Intent:

Because RC5 is not only outdated, but obsolete, and because more and more electronic devices from Asian consumer electronics manufacturers are found in the home, it is time to develop an IR decoder that can 'understand' about 90% of IR remotes that are used in our daily life.

This article introduces 'IRMP' as "Infrared Multi Protocol Decoder" in detail. The counterpart, the IRSND IR encoder, can be found in this document.

Inhaltsverzeichnis

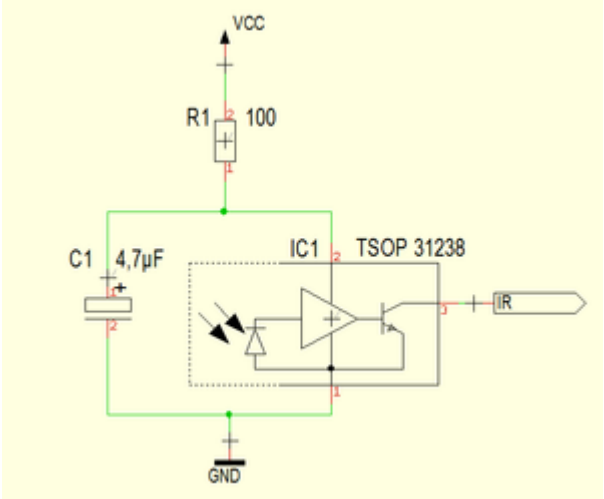
IRMP - Infrared Multi Protocol Decoder

Supported MCUs

IRMP runs on numerous MCU families:

AVR

- ATtiny87, ATtiny167
- ATtiny45, ATtiny85
- ATtiny44, ATtiny84
- ATmega8, ATmega16, ATmega32
- ATmega162
- ATmega164, ATmega324, ATmega644, ATmega644P, ATmega1284
- ATmega88, ATmega88P, ATmega168, ATmega168P, ATmega328P



Connection of a IR receiver module to MCU

XMega

- ATXmega128

PIC (CCS and XC8/C18 compiler)

- PIC12F1840
- PIC18F4520

STM32

- STM32F4xx (tested on STM32F401RE/F411RE Nucleo, STM32F4 Discovery)
- STM32F10x (tested on STM32F103C8T6 Mini Development Board)
- STM32 with HAL library **(NEW!)**

STM8

- STM8S103F3

TI Stellaris

- LM4F120 Launchpad (ARM Cortex M4)

ESP8266 (NEW!)

- ESP8266-EVB

TEENSY 3.0

- MK20DX256VLH7 (ARM Cortex-M4 72MHz)

MBED (NEW!)

- LPC1347 Cortex-M3 72 MHz
- LPC4088 (Embedded Artists)

ChibiOS HAL (NEW!)

- Various ARM Cortex MCUss, for example STM32, Kinetis, and NRF5
- Officially supported MCU series
- More MCU series, community supported

Supported IR Protocols

IRMP - the infrared remote decoder, which can decode several protocols at once - is capable of decoding the following protocols (in alphabetical order):

Supported Protocols

Protocol	Vendor
A1TVBOX	ADB (Advanced Digital Broadcast), e.g. A1 TV Box
APPLE	Apple
ACP24	Stiebel Eltron
B&O	Bang & Olufsen
BOSE	Bose
DENON	Denon, Sharp
FAN	FAN, remote for fans
FDC	FDC Keyboard
GRUNDIG	Grundig
NOKIA	Nokia, e.g. D-Box
IR60 (SDA2008)	various European vendors
JVC	JVC
KASEIKYO	Panasonic, Technics, Denon and other vendors which are members of the Japanese "Association for Electric Home Appliances" (AEHA).
KATHREIN	KATHREIN
LEGO	Lego
LGAIR	LG air conditioners
MATSUSHITA	Matsushita
MITSU_HEAVY	Mitsubishi air conditioners
NEC16	JVC, Daewoo
NEC42	JVC
MERLIN	MERLIN remote (Pollin article number: 620 185)
NEC	NEC, Yamaha, Canon, Tevion, Harman/Kardon, Hitachi, JVC, Pioneer, Toshiba, Xoro, Orion, generic and many other Asian vendors.
NETBOX	Netbox
Nikon	Nikon
NUBERT	Nubert, e.g. Subwoofer Systems
ORTEK	Ortek, Hama
PANASONIC	PANASONIC video projectors
PENTAX	PENTAX
RC5	Philips and other European vendors
RC6A	Philips, Kathrein and others, e.g. XBOX
RC6	Philips and other European vendors
RCCAR	RC Car: IR remote for RC toys
RCII	T+A **(NEW!)**
RECS80	Philips, Nokia, Thomson, Nordmende, Telefunken, Saba

RECS80EXT	Philips, Technisat, Thomson, Nordmende, Telefunken, Saba
RCMM	Fujitsu-Siemens e.g. Activy keyboard
ROOMBA	iRobot Roomba vacuum cleaner
S100	similar to RC5, but 14 instead of 13 bits and 56kHz modulation. Vendor unknown.
SAMSUNG32	Samsung
SAMSUNG48	various air conditioners
SAMSUNG	Samsung
RUWIDO	RUWIDO (e.g. T-Home Media Receiver, MERLIN keyboard(Pollin))
SIEMENS	Siemens, e.g. Gigaset M740AV
SIRCS	Sony
SPEAKER	Speaker systems like X-Tensions
TECHNICS	Technics
TELEFUNKEN	Telefunken
THOMSON	Thomson
VINCENT	Vincent

New:

Starting with version 3.2, IRMP can also decode 433 MHz RF radio protocols.

Supported RF Protocols

Protocol	Vendor
RF_GEN24	Generiv 24-bit format, e.g. Pollin 550666 radio-controlled receptacle
RF_X10	X10 PC RF remote control (Medion), Pollin 721815

Each of these protocols can be activated separately. If you want, you can activate all protocols. If you need only one protocol, you can disable all others. Only the code selected by the user will be compiled .

History

The IRMP source for the AVR and PIC MCUs was created as part of the Word Clock project.

Thread in Forum

Intention for an own IRMP article is the following thread in Projects&Code IRMP - Infrared Multi Protocol Decoder (in German language).

IR Protocols

Some vendors use their own proprietary protocols, such as Sony, Samsung, and Matsushita. Philips developed and used RC5.

RC5 was seen in Europe as *the* standard IR protocol which was adopted by many European vendors. Nowadays RC5 is practically not used and can be considered "dead". Although the successor RC6 is used in current European hardware, it is also used rarely.

Japanese vendors also tried to establish their own standard, the so called Kaseikyo (or "Japan") protocol. With a word length of 48 bits, it is more versatile. But it has not found wide use, though it is found in some appliances.

Nowadays the NEC protocol is used (also mainly in Japanese devices) in both premium and generic products. I estimate the market share at 80% for the NEC protocol. Nearly all remotes in my daily use utilize the NEC IR code. This starts with the TV set, continues with the DVD player and the notebook remote all the way to the generic multimedia hard drive, just to mention a few examples.

NEC protocol, RGB remote control, T->A: 9.14ms, A->B: 4.42ms, B->C: 660us

Coding methods

IRMP supports the following IR coding methods:

- Pulse Distance, typ. Example: NEC
- Pulse Width, typ. Example: Sony SIRCS
- Biphase (Manchester), typ. Example: Philips RC5, RC6
- Pulse Position (NRZ), typ. Example: Netbox
- Pulse Distance Width, typ. Example: Nubert

The pulses are modulated - usually at 36 kHz or 38 kHz - to reduce environmental influences such as indoor lighting or sunlight.

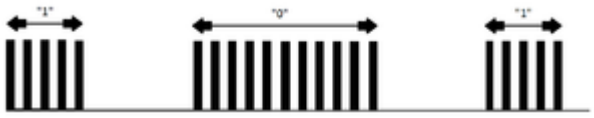
Pulse Distance

Pulse distance coding can be identified by the following rule:

- there is only **one pulse length** and there are **two different space lengths**

Pulse distance coding

Pulse Width

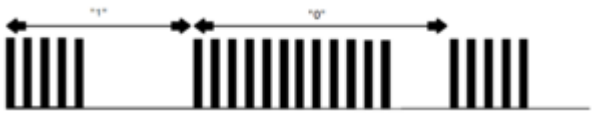


Pulse width coding

Pulse width coding can be identified by the following rule:

- there are **two different pulse lengths** and **only one space length**

Pulse Distance Width



Pulse distance width coding

This is a mix of pulse distance and pulse width coding. Often the sum of pulse and space length is constant. The rule is:

- there are **two different pulse lengths** and **two different space lengths.**

Biphase

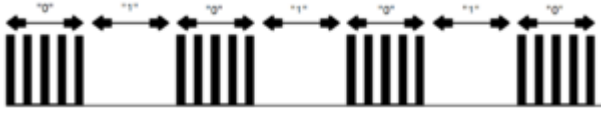
Biphase coding

In biphase coding the order of pulse and space gives the bit value. Therefore a biphase boding can be identified by this criteria:

- there is exactly **one** pulse and space length, as well as the **double** pulse/space length

Usually the length for the pulse and space are equal, meaning that the signal shape is symmetric. But IRMP also recognizes protocols which use different pulse and space lengths, for example the A1TVBOX protocol.

Pulse Position



Pulse position coding

Pulse position coding is known from commmon UARTs. Here, every bit has a fixed length. Depending on the value (0 or 1), it is a pulse or a space.

Typical criteria for a **pulse position protocol** are:

- there are **multiples** of a basic pulse/space length

A tabular listing of different IR protocols can be found here: IR Protocols in Detail. The specified timings are typical values. In some remotes they differ up to 40% in real life. Therefore IRMP uses minimum/maximum limits to be tolerant with the timing.

Protocol Detection

Most of the protocols IRMP decodes have something in common: they exhibit a start bit whose timing is unique.

According to this start bit timing, most protocols can be identified. IRMP measures the timing of the start bit and adjusts its timing tables "on-the-fly" to the discovered protocol. Subsequent bits can then be read sequentially without the need to first store a complete frame. Thus, IRMP does not wait to read a complete frame but starts decoding directly after detecting the first pulse.

If the start bit is not unique, IRMP proceeds in parallel with multiple possible protocols. For example with two candidate protocols, once plausible reasons disqualify one protocol, the other protocol is used.

Detection is implemented as an interrupt-driven state machine which is called at a frequency of typically 15000 times per second. Among others, the state machine has the following states:

- detect the first pulse length of the start bit
- detect the space length of the start bit
- detect the space length of the first data bit

After that, the pulse and space lengths of the start bit are known. Now all enabled protocols are searched for these lengths. If a protocol matches, the timing table for this protocol is loaded. Subsequent bits are checked against the timing table to ensure they conform to it.

The state machine continues with the following states:

- detect the spaces of the data bits
- detect the pulse length of the data bits
- check timing. If different, switch back to another valid IR protocol, otherwise return
- detect the stop bit, if present in the protocol
- check data for plausibility, like CRC or other redundancy bits
- convert data to device address and command
- detect code repetition by long key press, set corresponding flag

Indeed the state machine is even more complex because some protocols have no start bit (e.g. Denon) or have multiple start bits (4 in B&O) or have another sync bit within the frame (e.g. Samsung). These extra conditions are caught in the code by protocol-specific "special cases".

Switching to an other protocol can happen multiple times while receiving a frame, e.g. from NEC42 (42 bits) to NEC16 (8 bits + sync bit + 8 bits), if a premature sync bit is detected. Or from NEC/NEC42 (32/42 bits) to JVC (16 bits) if the stop bit occurs prematurely. It becomes difficult when, after detecting the start bit, two possible protocols use different coding methods, e.g. when the one protocol uses pulse distance coding and the other uses biphase coding (Manchester). In this case IRMP stores the necessary bits for both coding methods and later discards one or the other.

Furthermore, some remotes using particular protocols transmit repeat frames, either for redundancy (error detection) or for long key presses. Both kinds are detected by IRMP: error detection frames are detected by IRMP, but not passed to the application. Others are detected as long key presses and flagged by IRMP.

Download

Version 3.2.6, Date: 2021-01-27

Download stable version: Irmp.zip

Current development version of IRMP & IRSND:

- SVN link: SVN
- SVN browser: IRMP in SVN
- Download tarball: Tarball.

Download Arduino library: GitHub or use Arduino "*Tools / Manage Libraries...*" and search for IRMP.

You can see the history of the software changes here: Software History

License

IRMP is Open Source Software and is released under the GPL v2, or (at your option) any later version.

Source Code

The source code can be easily compiled for AVR MCUs by loading the project file irmp.aps in AVR Studio 4.

For other development environments it is simple to create a project or makefile. The source includes:

- irmp.c - IR decoder core
- irmpprotocols.h - all protocol definitions
- irmpsystem.h - target-independent definitions for AVR/PIC/STM32
- irmp.h - include file for the application

- irmpconfig.h - user configuration

Sample applications (main functions and timer configurations):

- irmp-main-avr.c - AVR
- irmp-main-avr-uart.c - AVR with UART output
- irmp-main-pic-xc8.c - PIC18F4520
- irmp-main-pic-12F1840.c - PIC12F1840
- irmp-main-stm32.c - STM32
- irmp-main-stellaris-arm.c - TI Stellaris LM4F120 Launchpad
- irmp-main-esp8266.c - ESP8266
- irmp-main-mbed.cpp - MBED
- examples/Arduino/Arduino.ino - Teensy 3.x
- irmp-main-chibios.c - ChibiOS

Important

include only irmp.h in your application:

```
#include "irmp.h"
```

All other include files are included within irmp.h. See also the sample application irmp-main-avr.c.

Furthermore, the preprocessor constant **F_CPU in project or makefile** must be defined. This should have at least the value of 8000000UL, processor speed should be at least 8 MHz. This applies to AVR targets and not for MCUs with PLL.

IRMP also runs on PIC processors. For the PIC-CCS compiler the necessary preprocessor defines are already set, such that irmp.c can be directly used in the CCS environment. Only a short interrupt service routine like

```
void  TIMER2_isr(void)
{
 irmp_ISR ();
}
```

must be added. The interrupt period time must be set to 66 µs (15 kHz).

For AVR processors you will find an example for the usage of IRMP in irmp-main-avr.c. The main things are the Timer initializing of the timer and the processing of received IR commands. The received protocol, the device address and the command will be output on the HW-UART.

For the Stellaris LM4F120 Launchpad from TI (ARM Cortex M4) is a propriate timer init function already integrated in irmp-main-avr.c.

IRMP can be used also with STM32 microcontrollers.

Another new implementation is available on the mbed platform.

avr-gcc Optimizations

From version 4.7.x of avr-gcc, the LTO option can be used to make the call of the external function irmp_ISR() from the main ISR more efficient. This improves the performance of the ISR a little.

Add the following compiler and linker options:

- additional compiler option: -flto
- additional linker options: -flto -Os

If you forget the additional linker option -Os, the binary will be significantly larger as it will not be optimized further. Also, the option -flto must be passed to the linker, otherwise link time optimization will not work.

Configuration

IRMP is configured by parameters in irmpconfig.h:

- number of interrupts per second
- supported IR protocols
- hardware pin for IR receiver
- IR logging

Settings in irmpconfig.h

IRMP will decode all protocols listed above in one ISR. For this, some settings are needed. These are set in irmpconfig.h.

F_INTERRUPTS

Number of interrupts per second. Should be set to a value from 10000 to 20000. The higher the value, the better the resolution and therefore the quality of detection. But a higher interrupt rate means also higher CPU load. A value of 15000 is usually a good compromise.

Default value:

```
#define F_INTERRUPTS                            15000      // interrupts per second
```

On AVR controllers, the irmp-main-avr.c example uses Timer1 with 16 bits of resolution. If for any reasons Timer1 is not available, you can also use Timer2 with 8 bits of resolution.

In that case, configure Timer2 as follows:

```
OCR2   =  (uint8_t) ((F_CPU / F_INTERRUPTS) / 8) - 1 + 0.5);   // Compare Register
OCR2
TCCR2  = (1 << WGM21) | (1 << CS21);                          // CTC Mode, prescaler
= 8
TIMSK  = 1 << OCIE2;                                          // enable Timer2
interrupt
```

The above example is valid for ATmega88/ATmega168/ATmega328. For other AVR MCUs check the datasheet.

You must not forget to change the ISR to Timer2 as well:

```
ISR(TIMER2_COMP_vect)
{
  (void) irmp_ISR();
}
```

IRMP_SUPPORT_xxx_PROTOCOL

Here you can select which protocols to enable in IRMP. Common protocols are emabled by default.

To enable additional protocols or disable others to save memory, set the corresponding values in irmpconfig.h.

```
// typical protocols, disable here!             Enable  Remarks
F_INTERRUPTS            Program Space
#define IRMP_SUPPORT_SIRCS_PROTOCOL             1       // Sony SIRCS           >=
10000                   ~150 bytes
#define IRMP_SUPPORT_NEC_PROTOCOL               1       // NEC + APPLE          >=
10000                   ~300 bytes
#define IRMP_SUPPORT_SAMSUNG_PROTOCOL           1       // Samsung + Samsung32  >=
10000                   ~300 bytes
#define IRMP_SUPPORT_MATSUSHITA_PROTOCOL        1       // Matsushita           >=
10000                    ~50 bytes
#define IRMP_SUPPORT_KASEIKYO_PROTOCOL          1       // Kaseikyo             >=
10000                   ~250 bytes

// more protocols, enable here!                 Enable  Remarks
F_INTERRUPTS            Program Space
#define IRMP_SUPPORT_DENON_PROTOCOL             0       // DENON, Sharp         >=
10000                   ~250 bytes
#define IRMP_SUPPORT_RC5_PROTOCOL               0       // RC5                  >=
10000                   ~250 bytes
#define IRMP_SUPPORT_RC6_PROTOCOL               0       // RC6 & RC6A           >=
10000                   ~250 bytes
#define IRMP_SUPPORT_JVC_PROTOCOL               0       // JVC                  >=
10000                   ~150 bytes
#define IRMP_SUPPORT_NEC16_PROTOCOL             0       // NEC16                >=
10000                   ~100 bytes
#define IRMP_SUPPORT_NEC42_PROTOCOL             0       // NEC42                >=
10000                   ~300 bytes
#define IRMP_SUPPORT_IR60_PROTOCOL              0       // IR60 (SDA2008)       >=
10000                   ~300 bytes
#define IRMP_SUPPORT_GRUNDIG_PROTOCOL           0       // Grundig              >=
10000                   ~300 bytes
#define IRMP_SUPPORT_SIEMENS_PROTOCOL           0       // Siemens Gigaset      >=
15000                   ~550 bytes
#define IRMP_SUPPORT_NOKIA_PROTOCOL             0       // Nokia                >=
10000                   ~300 bytes

// exotic protocols, enable here!               Enable  Remarks
F_INTERRUPTS            Program Space
#define IRMP_SUPPORT_BOSE_PROTOCOL              0       // BOSE                 >=
10000                   ~150 bytes
#define IRMP_SUPPORT_KATHREIN_PROTOCOL          0       // Kathrein             >=
10000                   ~200 bytes
#define IRMP_SUPPORT_NUBERT_PROTOCOL            0       // NUBERT               >=
```

```
10000                   ~50 bytes
#define IRMP_SUPPORT_BANG_OLUFSEN_PROTOCOL      0       // Bang & Olufsen       >=
10000                   ~200 bytes
#define IRMP_SUPPORT_RECS80_PROTOCOL            0       // RECS80 (SAA3004)     >=
15000                   ~50 bytes
#define IRMP_SUPPORT_RECS80EXT_PROTOCOL         0       // RECS80EXT (SAA3008)  >=
15000                   ~50 bytes
#define IRMP_SUPPORT_THOMSON_PROTOCOL           0       // Thomson              >=
10000                   ~250 bytes
#define IRMP_SUPPORT_NIKON_PROTOCOL             0       // Nikon camera         >=
10000                   ~250 bytes
#define IRMP_SUPPORT_NETBOX_PROTOCOL            0       // Netbox keyboard      >=
10000                   ~400 bytes (PROTOTYPE!)
#define IRMP_SUPPORT_ORTEK_PROTOCOL             0       // ORTEK (Hama)         >=
10000                   ~150 bytes
#define IRMP_SUPPORT_TELEFUNKEN_PROTOCOL        0       // Telefunken 1560      >=
10000                   ~150 bytes
#define IRMP_SUPPORT_FDC_PROTOCOL               0       // FDC3402 keyboard     >=
10000 (better 15000)  ~150 bytes (~400 in combination with RC5)
#define IRMP_SUPPORT_RCCAR_PROTOCOL             0       // RC Car               >=
10000 (better 15000)  ~150 bytes (~500 in combination with RC5)
#define IRMP_SUPPORT_ROOMBA_PROTOCOL            0       // iRobot Roomba        >=
10000                   ~150 bytes
#define IRMP_SUPPORT_RUWIDO_PROTOCOL            0       // RUWIDO, T-Home       >=
15000                   ~550 bytes
#define IRMP_SUPPORT_A1TVBOX_PROTOCOL           0       // A1 TV BOX            >=
15000 (better 20000)  ~300 bytes
#define IRMP_SUPPORT_LEGO_PROTOCOL              0       // LEGO Power RC        >=
20000                   ~150 bytes
#define IRMP_SUPPORT_RCMM_PROTOCOL              0       // RCMM 12, 24, or 32    >=
20000                   ~150 bytes
```

Each IR protocol enabled in IRMP consumes the amount of code noted above. Here you can apply optimizations: for example, the modulation frequency of 455 kHz for the B&O protocol is far away from the frequencies that are used by other protocols. This usually requires a different IR receiver, so without that, you can disable these protocols. For example, you cannot receive the B&O protocol (455kHz) with a TSOP1738/TSOP31238.

Additionally, the SIEMENS/FDC/RCCAR can only be detected reliably at a frequency of 15 kHz or higher. For LEGO, 20 kHz is needed. To use these protocols, you must modify F_INTERRUPTS. Otherwise, during compilation you will get a warning and the corresponding protocols will automatically be disabled.

IRMP_PORT_LETTER + IRMP_BIT_NUMBER

This constant defines the pin where the IR receiver is connected.

Default value is PORT B6:

```
/*---------------------------------------------------------------------------
 * Change hardware pin here for ATMEL AVR
 *---------------------------------------------------------------------------
 */
#if defined (ATMEL_AVR)                         // use PB6 as IR input on AVR
#  define IRMP_PORT_LETTER                      B
#  define IRMP_BIT_NUMBER                       6
```

These two values must match your hardware configuration.

This applies also to STM32 MCUs:

```
/*---------------------------------------------------------------------------
 * Change hardware pin here for ARM STM32
 *---------------------------------------------------------------------------
 */
#elif defined (ARM_STM32)                       // use C13 as IR input on STM32
#  define IRMP_PORT_LETTER                      C
#  define IRMP_BIT_NUMBER                       13
```

When using STM32 HAL library, define the constants IRSND_Transmit_GPIO_Port and IRSND_Transmit_Pin in STM32Cube (Main.h). In this case, it is not necessary to change the constants in irmpconfig.h:

```
/*---------------------------------------------------------------------------------------------------------------
 * ARM STM32 with HAL section - don't change here, define IRSND_Transmit_GPIO_Port & IRSND_Transmit_Pin in STM32Cube (Main.h)
 *---------------------------------------------------------------------------------------------------------------
 */
#elif defined (ARM_STM32_HAL)                                                   // IRSND_Transmit_GPIO_Port & IRSND_Transmit_Pin must be defined in STM32Cube
#  define IRSND_PORT_LETTER                     IRSND_Transmit_GPIO_Port//Port of Transmit PWM Pin e.g.
#  define IRSND_BIT_NUMBER                      IRSND_Transmit_Pin      //Pim of Transmit PWM Pin e.g.
#  define IRSND_TIMER_HANDLER                   htim2                   //Handler of Timer e.g. htim (see tim.h)
#  define IRSND_TIMER_CHANNEL_NUMBER            TIM_CHANNEL_2           //Channel of the used Timer PWM Pin e.g. TIM_CHANNEL_2
#  define IRSND_TIMER_SPEED_APBX                64000000                //Speed of the corresponding APBx. (see STM32CubeMX: Clock Configuration)
```

And the corresponding section for STM8 MCUs:

```
/*---------------------------------------------------------------------------
 * Change hardware pin here for STM8
 *---------------------------------------------------------------------------
 */
#elif defined (SDCC_STM8)                       // use PA1 as IR input on STM8
#  define IRMP_PORT_LETTER                      A
#  define IRMP_BIT_NUMBER                       1
```

For PIC microcontrollers only the constant **IRMP_PIN** needs to be changed - depending on the compiler:

```
/*---------------------------------------------------------------------------
 * Change hardware pin here for PIC C18 compiler
 *---------------------------------------------------------------------------
 */
#elif defined (PIC_C18)                         // use RB4 as IR input on PIC
#  define IRMP_PIN                              PORTBbits.RB4

/*---------------------------------------------------------------------------
 * Change hardware pin here for PIC CCS compiler
 *---------------------------------------------------------------------------
 */
#elif defined (PIC_CCS)                         // use PB4 as IR input on PIC
#  define IRMP_PIN                              PIN_B4
```

When using ChibiOS HAL, define a pin with the name **IR_IN** in your board config (board.chcfg) of ChibiOS and regenerate the board files. To use another name for the pin, edit the constant **IRMP_PIN** in irmpconfig.h. Use the name of the pin from the board config and prefix it with "LINE_", as IRMP is using the "line" variant of the PAL interface:

```
/*---------------------------------------------------------------------------
-------------------------------------------------------------
 * Change hardware pin here for ChibiOS HAL
 *---------------------------------------------------------------------------
-------------------------------------------------------------
 */
#elif defined(_CHIBIOS_HAL_)
#  define IRMP_PIN                              LINE_IR_IN              // use pin
names as defined in the board config file, prefixed with "LINE_"
```

IRMP_HIGH_ACTIVE

Default value:

```
#  define IRMP_HIGH_ACTIVE                     0                       // set to 1
if you use a RF receiver!
```

Most RF receivers use active-high signals, so when using an RF receiver instead of an IR sensor, set this value to 1.

NEU:

IRMP_ENABLE_RELEASE_DETECTION

Default value:

```
#  define IRMP_ENABLE_RELEASE_DETECTION        0                       //  enable
detection of key releases
```

Set this value to 1 to enable detection of button release events. The function irmp_get_data() then sets the IRMP_FLAG_RELEASE bit in the struct member irmp_data.flags once code transmission ends. See the example in the section **Debouncing**.

IRMP_USE_CALLBACK

Default value:

```
#define IRMP_USE_CALLBACK                     0        // flag: 0 = don't use
callbacks, 1 = use callbacks, default is 0
```

When you turn on callbacks, any level change at the input causes the callback function to be called. This can be used to visualize incoming IR signals by driving another output pin.

Here is an example:

```
#define LED_PORT PORTD                                  // LED at PD6
#define LED_DDR  DDRD
#define LED_PIN  6
```

```
/*---------------------------------------------------------------------------
-----------------------------------
 * Called (back) from IRMP module
 * This example switches a LED (which is connected to Vcc)
 *---------------------------------------------------------------------------
-----------------------------------
 */
void
led_callback (uint_fast8_t on)
{
    if (on)
    {
       LED_PORT &= ~(1 << LED_PIN);
    }
    else
    {
       LED_PORT |= (1 << LED_PIN);
    }
}

int
main ()
{
    ...
    irmp_init ();

    LED_DDR |= (1 << LED_PIN);          // LED pin to output
    LED_PORT |= (1 << LED_PIN);         // switch LED off (active low)
    irmp_set_callback_ptr (led_callback);

    sei ();
    ...
}
```

IRMP_USE_IDLE_CALL

Normally the irmp_ISR() function is called continuously at the F_INTERRUPTS (10-20kHz) frequency. The microcontroller can rarely enter an energy-saving sleep mode, or must constantly wake up from it. If power consumption is important, e.g. on battery power, this approach is not optimal.

If *IRMP_USE_IDLE_CALL'* is enabled, IRMP detects if no IR transmission is ongoing and then calls the function **irmp_idle()**. This is microcontroller-specific and must be provided and linked by the user. The microcontroller can then be put to sleep while there is no ongoing transmission, thus reducing energy consumption.

It is recommended to deactivate the timer interrupt in irmp_idle() and to activate a pin change interrupt instead. Then the microcontroller can be put to sleep. When a falling edge is detected on the IR input pin, the pin change interrupt is disabled, the timer is reenabled and irmp_ISR() is called immediately. You can find an example for the use of irmp_idle() in irmp-main-chibios.c.

Using IRMP purely with pin change interrupts and without timer interrupts is not supported.

IRMP_USE_EVENT

When using IRMP with ChibiOS/RT or ChibiOS/NIL, you can use their Event module to wake a thread as soon as new IR data is received and decoded.

Set the **IRMP_USE_EVENT** constant in irmpconfig.h to 1 to enable this. **IRMP_EVENT_BIT** definies the value in the Event bitmask that should symbolize the IRMP event. Use **IRMP_EVENT_THREAD_PTR** to define the variable name of the thread pointer that the event is sent to.

Change irmpconfig.h like this:

```
/*-----------------------------------------------------------------------------------------
-----------------------------------------
 * Use ChibiOS Events to signal that valid IR data was received
 *-----------------------------------------------------------------------------------------
-----------------------------------------
 */
#if defined(_CHIBIOS_RT_) || defined(_CHIBIOS_NIL_)

#  ifndef IRMP_USE_EVENT
#    define IRMP_USE_EVENT                      1                   // 1: use
event. 0: do not. default is 0
#  endif

#  if IRMP_USE_EVENT == 1 && !defined(IRMP_EVENT_BIT)
#    define IRMP_EVENT_BIT                      1                   // event flag
or bit to send
#  endif
#  if IRMP_USE_EVENT == 1 && !defined(IRMP_EVENT_THREAD_PTR)
#    define IRMP_EVENT_THREAD_PTR               ir_receive_thread_p   // pointer to
the thread to send the event to
extern thread_t *IRMP_EVENT_THREAD_PTR;                               // the pointer
must be defined and initialized elsewhere
#  endif

#endif // _CHIBIOS_RT_ || _CHIBIOS_NIL_
```

Now you can use the event in your ChibiOS project like this:

```
thread_t *ir_receive_thread_p = NULL;

static THD_FUNCTION(IRThread, arg)
{
    ir_receive_thread_p = chThdGetSelfX();
    [...]
    while (true)
    {
        // wait for event sent from irmp_ISR
        chEvtWaitAnyTimeout(ALL_EVENTS,TIME_INFINITE);

        if (irmp_get_data (&irmp_data))
            // use data in irmp_data
```

IRMP_LOGGING

With IRMP_LOGGING the logging of received IR frames can be turned on.

Default value:

```
#define IRMP_LOGGING                    0       // 1: log IR signal (scan),
0: do not. default is 0
```

Further documentation can be found here: Scanning Unknown IR Protocols.

Using IRMP

The protocols supported by IRMP use partly variable, partly fixed bit lengths from 2 up to 48 bits. These are described by preprocessor defines.

IRMP separates these IR frames into 3 sections:

1. protocol ID
2. address or vendor code
3. command

With the function

```
irmp_get_data (IRMP_DATA * irmp_data_p)
```

you can recall a decoded message. The return value is 1 if a message has been received, otherwise it is 0. In the first case the struct members

```
irmp_data_p->protocol (8 Bit)
irmp_data_p->address (16 Bit)
irmp_data_p->command (16 Bit)
irmp_data_p->flags (8 Bit)
```

contain valid information.

That means that ultimately, you have three values (protocol, address and command) that can be evaluated with an if or switch statement. Here is a sample decoder which listens for keys 1-9 on a remote:

```
IRMP_DATA irmp_data;

if (irmp_get_data (&irmp_data))
{
   if (irmp_data.protocol == IRMP_NEC_PROTOCOL &&     // NEC protocol
       irmp_data.address == 0x1234)                   // Address 0x1234
   {
      switch (irmp_data.command)
      {
         case 0x0001: key1_pressed(); break;          // Key 1
         case 0x0002: key2_pressed(); break;          // Key 2
         ...
         case 0x0009: key9_pressed(); break;          // Key 9
      }
   }
}
```

Here are possible constants for irmp_data.protocol, see also irmpprotocols.h:

```
#define IRMP_SIRCS_PROTOCOL                 1           // Sony
#define IRMP_NEC_PROTOCOL                   2           // NEC, Pioneer, JVC,
Toshiba, generic, etc.
#define IRMP_SAMSUNG_PROTOCOL               3           // Samsung
```

```
#define IRMP_MATSUSHITA_PROTOCOL                4               // Matsushita
#define IRMP_KASEIKYO_PROTOCOL                  5               // Kaseikyo
(Panasonic, etc.)
#define IRMP_RECS80_PROTOCOL                    6               // Philips, Thomson,
Nordmende, Telefunken, Saba
#define IRMP_RC5_PROTOCOL                       7               // Philips etc
#define IRMP_DENON_PROTOCOL                     8               // Denon, Sharp
#define IRMP_RC6_PROTOCOL                       9               // Philips etc
#define IRMP_SAMSUNG32_PROTOCOL                 10              // Samsung32: no sync
pulse at bit 16, length 32 instead of 37
#define IRMP_APPLE_PROTOCOL                     11              // Apple, very
similar to NEC
#define IRMP_RECS80EXT_PROTOCOL                 12              // Philips,
Technisat, Thomson, Nordmende, Telefunken, Saba
#define IRMP_NUBERT_PROTOCOL                    13              // Nubert
#define IRMP_BANG_OLUFSEN_PROTOCOL              14              // Bang & Olufsen
#define IRMP_GRUNDIG_PROTOCOL                   15              // Grundig
#define IRMP_NOKIA_PROTOCOL                     16              // Nokia
#define IRMP_SIEMENS_PROTOCOL                   17              // Siemens, e.g.
Gigaset
#define IRMP_FDC_PROTOCOL                       18              // FDC keyboard
#define IRMP_RCCAR_PROTOCOL                     19              // RC Car
#define IRMP_JVC_PROTOCOL                       20              // JVC (NEC with 16
bits)
#define IRMP_RC6A_PROTOCOL                      21              // RC6A, e.g.
Kathrein, XBOX
#define IRMP_NIKON_PROTOCOL                     22              // Nikon
#define IRMP_RUWIDO_PROTOCOL                    23              // Ruwido, e.g. T-
Home Media Receiver
#define IRMP_IR60_PROTOCOL                      24              // IR60 (SDA2008)
#define IRMP_KATHREIN_PROTOCOL                  25              // Kathrein
#define IRMP_NETBOX_PROTOCOL                    26              // Netbox keyboard
(bitserial)
#define IRMP_NEC16_PROTOCOL                     27              // NEC with 16 bits
(incl. sync)
#define IRMP_NEC42_PROTOCOL                     28              // NEC with 42 bits
#define IRMP_LEGO_PROTOCOL                      29              // LEGO Power
Functions RC
#define IRMP_THOMSON_PROTOCOL                   30              // Thomson
#define IRMP_BOSE_PROTOCOL                      31              // BOSE
#define IRMP_A1TVBOX_PROTOCOL                   32              // A1 TV Box
#define IRMP_ORTEK_PROTOCOL                     33              // ORTEK - Hama
#define IRMP_TELEFUNKEN_PROTOCOL                34              // Telefunken (1560)
#define IRMP_ROOMBA_PROTOCOL                    35              // iRobot Roomba
vacuum cleaner
#define IRMP_RCMM32_PROTOCOL                    36              // Fujitsu-Siemens
(Activy remote control)
#define IRMP_RCMM24_PROTOCOL                    37              // Fujitsu-Siemens
(Activy keyboard)
#define IRMP_RCMM12_PROTOCOL                    38              // Fujitsu-Siemens
(Activy keyboard)
#define IRMP_SPEAKER_PROTOCOL                   39              // Another
loudspeaker protocol, similar to Nubert
#define IRMP_LGAIR_PROTOCOL                     40              // LG air conditioner
#define IRMP_SAMSUNG48_PROTOCOL                 41              // air conditioner
with Samsung protocol (48 bits)
#define IRMP_MERLIN_PROTOCOL                    42              // Merlin (Pollin 620
185)
#define IRMP_PENTAX_PROTOCOL                    43              // Pentax camera
#define IRMP_FAN_PROTOCOL                       44              // FAN (ventilator),
very similar to NUBERT, but last bit is data bit instead of stop bit
#define IRMP_S100_PROTOCOL                      45              // very similar to
RC5, but 14 instead of 13 data bits
```

```
#define IRMP_ACP24_PROTOCOL                     46              // Stiebel Eltron
ACP24 air conditioner
#define IRMP_TECHNICS_PROTOCOL                  47              // Technics, similar
to Matsushita, but 22 instead of 24 bits
#define IRMP_PANASONIC_PROTOCOL                 48              // Panasonic (video
projector), start bits similar to KASEIKYO
#define IRMP_MITSU_HEAVY_PROTOCOL               49              // Mitsubishi heavy
air conditioner, similar timing to Panasonic video projector
#define IRMP_VINCENT_PROTOCOL                   50              // Vincent
#define IRMP_SAMSUNGAH_PROTOCOL                 51              // Samsung AH
#define IRMP_IRMP16_PROTOCOL                    52              // IRMP specific
protocol for data transfer, e.g. between two microcontrollers via IR
#define IRMP_GREE_PROTOCOL                      53              // Gree climate
#define IRMP_RCII_PROTOCOL                      54              // RC II Infrared
Remote Control Protocol for FM8
#define IRMP_METZ_PROTOCOL                      55              // METZ
#define IRMP_ONKYO_PROTOCOL                     56              // Onkyo
```

The values of address and the command code of an unknown remote can be received and printed to UART or LCD. Then these values can be hard-coded in your decoder routine. Or you can write a learning routine, where you press keys to store the code into EEPROM. A sample for this can be found in Lernfähige IR-Fernbedienung mit IRMP.

Another example main function is included in the zip file, showing also the timer initialization.

Debouncing

To distinguish between a long key press or a single press, the IRMP_FLAG_REPETITION bit is provided. It is set in the struct member **flags** when a key on the remote is held, causing the same command to be repeated within a short period of time.

Example:

```
if (irmp_data.flags & IRMP_FLAG_REPETITION)
{
  // long key press
  // either:
  //   ignore the (repeated) key
  // or:
  //   use this information for a repeat function
}
else
{
  // key was pressed again
}
```

This can be used to debounce the keys 0-9 by ignoring commands with the IRMP_FLAG_REPETITION bit set. For keys like 'VOLUME+' or 'VOLUME-' using the repetition can be useful, for example to fade a LED.

If you want to decode only single keys, you can reduce the block above to:

```
if (! (irmp_data.flags & IRMP_FLAG_REPETITION))
{
  // New key
  // ACTION!
}
```

NEW:

From version 3.2.2, key releases can be detected. In this case, the IRMP_FLAG_RELEASE flag is set once the remote control has ceased sending IR or RF frames.

Example:

```
    IRMP_DATA irmp_data;

    while (1)
    {
        if (irmp_get_data (&irmp_data))
        {
            if (irmp_data.protocol == NEC_PROTOCOL && irmp_data.address == 0x1234)
            {
                if (irmp_data.command == 0x42 && irmp_data.flags == 0x00) // First
frame, flags not set
                {
                    motor_on ();
                }
                else if (irmp_data.flags & IRMP_FLAG_RELEASE)             // Key is
released
                {
                    motor_off ();
                }
            }
        }
    }
```

In the above example, a motor is turned on when a specific button on the remote control is pressed. The motor will not stop again until you release the button.

Important when evaluating IRMP_FLAG_RELEASE:

You must not rely on irmp_data.command to still contain the original command code (0x42 here). There are remote controls (e.g. RF remotes for remote controlled receptacles) which send a special key release code when the key is released. Simply check that the irmp_data.address matches before evaluating the flag.

This feature must be enabled explicitly in irmpconfig.h by changing the configuration variable IRMP_ENABLE_RELEASE_DETECTION.

Principles of Operation

The "workhorse" of IRMP is the interrupt service routine irmp_ISR() which should be called 15000 times per second. When using a different rate, the constant F_INTERRUPTS in irmpconfig.h needs to be modified accordingly.

First, irmp_ISR() detects the length and the type of the start bit(s) and uses this to determine the protocol in use. As soon as the protocol is identified, subsequent bits are parametrized to read them efficiently until the IR transmission is complete.

To cut off critics:

I know that the ISR is quite large. But since it behaves like a state machine, the effective executed code per cycle is relatively small. As long as the input is "dark" (and that is the case most of the time ;-)) the spent time is vanishingly short. In the WordClock project for example, 8 ISRs are called with the same timer, of which irmp_ISR() just one of many. With a MCU clock of at least 8 MHz, no timing problems occured. Consequently, I see no problem with the length of the ISR.

A crystal is not mandatory, it works well with the internal AVR oscillator. Remember to set the correct fuses for the CPU to run at 8 MHz, check irmp-main-avr.c for correct values for an ATMEGA88.

Scanning Unknown IR Protocols

To enable logging in IRMP, modify the value of IRMP_LOGGING to 1 of irmpconfig.h on the line

```
#define IRMP_LOGGING    0   // 1: log IR signal (scan), 0: do not (default)
```

When logging is enabled, the bright and dark phases are sent via UART at 9600 bits/s: 1=dark, 0=bright. The constants in the functions uart_init() and uart_putc() may need to be modified, depending on the AVR MCU used.

Note: for PIC microcontrollers there is a dedicated logging module named irmpextlog.c. This makes it possible to log via USB. This does not apply to the AVR version

By capturing these protocol scans with a terminal program and saving them to a text file, you can use these files to analyze the frames in order to add new protocols to IRMP - see next chapter.

If you have a remote control that is not supported by IRMP, you can send me (ukw) the scan files. Then I can check if the protocol is compatible with the IRMP model and modify the source code if applicable.

IRMP under Linux and Windows

Compilation

irmp.c can be compiled under Linux for testing IR scans in textfiles. In the subdirectory 'IR-Data' you will find such files that you can use as input files for IRMP.

To compile IRMP, enter:

```
make -f makefile.lnx
```

This will generate 3 IRMP versions:

- irmp-10kHz: Version for 10kHz scans
- irmp-15kHz: Version for 15kHz scans
- irmp-20kHz: Version for 20kHz scans

Starting IRMP

The calling syntax is:

```
./irmp-nnkHz [-l|-p|-a|-v] < scan-file
```

Options are mutually exclusive, so only one option per call is valid:

Option:

```
  -l  List              print a list with pulses and pauses
  -a analyze            analyse the pulses/pauses and write a "spectrum"
in ASCII format
  -v verbose            verbose output
  -p  Print Timings     print a timing table for all protocols
```

Samples:

Normal Output

```
./irmp-10kHz < IR-Data/orion_vcr_07660BM070.txt
```

```
-----------------------------------------------------------------------------
# Taste 1
00000001110111101000000001111111 p =  2, a = 0x7b80, c = 0x0001, f = 0x00
-----------------------------------------------------------------------------
# Taste 2
00000001110111100100000010111111 p =  2, a = 0x7b80, c = 0x0002, f = 0x00
-----------------------------------------------------------------------------
# Taste 3
00000001110111101100000000111111 p =  2, a = 0x7b80, c = 0x0003, f = 0x00
-----------------------------------------------------------------------------
# Taste 4
00000001110111100010000011011111 p =  2, a = 0x7b80, c = 0x0004, f = 0x00
-----------------------------------------------------------------------------
...
```

Output Lists

```
./irmp-10kHz -l < IR-Data/orion_vcr_07660BM070.txt
```

```
# Taste 1
pulse: 91 pause: 44
pulse: 6 pause: 5
pulse: 6 pause: 6
pulse: 6 pause: 5
pulse: 6 pause: 5
pulse: 6 pause: 5
pulse: 6 pause: 6
pulse: 6 pause: 5
pulse: 6 pause: 16
...
```

Analysis

```
./irmp-10kHz -a < IR-Data/orion_vcr_07660BM070.txt
```

```
-----------------------------------------------------------------------------
START PULSES:
 90 o 1
```

```
 91 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 33
 92 ooo 2
pulse avg: 91.0=9102.8 us, min: 90=9000.0 us, max: 92=9200.0 us, tol:  1.1%
--------------------------------------------------------------------------------
START PAUSES:
 43 oo 1
 44 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 25
 45 oooooooooooooooooooooo 10
pause avg: 44.2=4425.0 us, min: 43=4300.0 us, max: 45=4500.0 us, tol:  2.8%
--------------------------------------------------------------------------------
PULSES:
  5 o 17
  6 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 562
  7 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 609
pulse avg:  6.5= 649.8 us, min:  5= 500.0 us, max:  7= 700.0 us, tol: 23.1%
--------------------------------------------------------------------------------
PAUSES:
  4 oooooooooooooooooooo 169
  5 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 412
  6 oooo 31
pause avg:  4.8= 477.5 us, min:  4= 400.0 us, max:  6= 600.0 us, tol: 25.7%
 15 oooooo 43
 16 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 425
 17 ooooooooo 72
pause avg: 16.1=1605.4 us, min: 15=1500.0 us, max: 17=1700.0 us, tol:  6.6%
--------------------------------------------------------------------------------
```

Here you see the measured times of all pulses and pauses (spaces) as horizontal bar graphs, whose distributions are not ideal bell curves due to the ASCII formatting. The narrower the measured peaks, the better the timing of the remote.

The above output can be read as:

- the start bit has a pulse length between 9000 and 9200 µs, on average 9102 µs. The deviation from this average is about 1.1%
- the start bit has a space length between 4300 and 4500 µs, the average is 4424 µs. The error is about 2.8%.
- the pulse length of a data bit is between 500 and 700 µs, on average 650 µs, the error is (quite large!) 23.1%

Further there are two more spaces of different lengths (for bits 0 and 1). Reading these is left as an exercise to the reader. ;-)

Verbose Output

```
./irmp-10kHz -v < IR-Data/orion_vcr_07660BM070.txt
```

```
--------------------------------------------------------------------------------
# 1 - IR-cmd: 0x0001
   0.200ms [starting pulse]
  13.700ms [start-bit: pulse = 91, pause = 44]
protocol = NEC, start bit timings: pulse:  62 - 118, pause:  30 -  60
pulse_1:   3 -   8
pause_1:  11 -  23
pulse_0:   3 -   8
pause_0:   3 -   8
```

```
  command_offset: 16
  command_len:     16
  complete_len:    32
  stop_bit:         1
    14.800ms [bit  0: pulse =   6, pause =   5] 0
    16.000ms [bit  1: pulse =   6, pause =   6] 0
    17.100ms [bit  2: pulse =   6, pause =   5] 0
    18.200ms [bit  3: pulse =   6, pause =   5] 0
    19.300ms [bit  4: pulse =   6, pause =   5] 0
    20.500ms [bit  5: pulse =   6, pause =   6] 0
    21.600ms [bit  6: pulse =   6, pause =   5] 0
    23.800ms [bit  7: pulse =   6, pause =  16] 1
    26.100ms [bit  8: pulse =   6, pause =  17] 1
    28.300ms [bit  9: pulse =   6, pause =  16] 1
    29.500ms [bit 10: pulse =   6, pause =   6] 0
    31.700ms [bit 11: pulse =   6, pause =  16] 1
    34.000ms [bit 12: pulse =   6, pause =  17] 1
    36.200ms [bit 13: pulse =   6, pause =  16] 1
    38.500ms [bit 14: pulse =   6, pause =  17] 1
    39.600ms [bit 15: pulse =   6, pause =   5] 0
    41.900ms [bit 16: pulse =   6, pause =  17] 1
    43.000ms [bit 17: pulse =   6, pause =   5] 0
    44.100ms [bit 18: pulse =   6, pause =   5] 0
    45.200ms [bit 19: pulse =   6, pause =   5] 0
    46.400ms [bit 20: pulse =   7, pause =   5] 0
    47.500ms [bit 21: pulse =   6, pause =   5] 0
    48.600ms [bit 22: pulse =   6, pause =   5] 0
    49.800ms [bit 23: pulse =   6, pause =   6] 0
    50.900ms [bit 24: pulse =   5, pause =   6] 0
    53.100ms [bit 25: pulse =   6, pause =  16] 1
    55.400ms [bit 26: pulse =   6, pause =  17] 1
    57.600ms [bit 27: pulse =   6, pause =  16] 1
    59.900ms [bit 28: pulse =   6, pause =  17] 1
    62.100ms [bit 29: pulse =   6, pause =  16] 1
    64.400ms [bit 30: pulse =   6, pause =  17] 1
    66.700ms [bit 31: pulse =   6, pause =  17] 1
  stop bit detected
    67.300ms code detected, length = 32
    67.300ms p =  2, a = 0x7b80, c = 0x0001, f = 0x00
  ---------------------------------------------------------------------------
```

Starting under Windows

IRMP can be used under Windows as well:

- start command line console
- change to directory 'irmp'
- enter:

```
irmp-10kHz.exe < IR-Data\rc5x.txt
```

The same options apply as for the Linux version.

Long Output

As some output is very long, it is recommended to redirect the output to a file or filter for paging:

Linux:

```
./irmp-10kHz < IR-Data/rc5x.txt | less
```

Windows:

```
irmp-10kHz.exe < IR-Data\rc5x.txt | more
```

Remote Controls

VCR|-

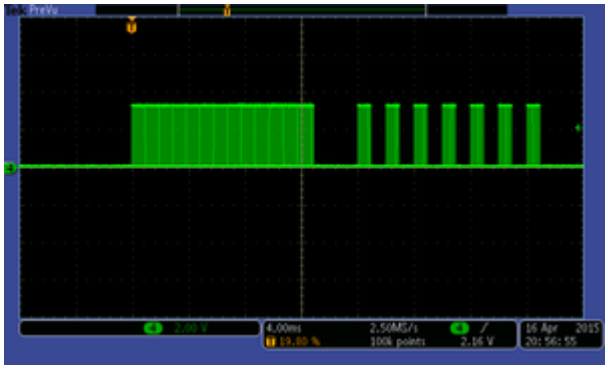
Protocol	Name	Device	Device Address
NEC	Toshiba CT-9859	TV	0x5F40
	Elta 8848 MP 4	DVD player	0x7F00
	AS-218	Askey TV-View CHP03X (TV tuner card)	0x3B86
	Cyberhome ???	Cyberhome DVD player	0x6D72
	WD TV Live	Western Digital Multimediaplayer	0x1F30
	Canon WL-DC100	Kamera Canon PowerShot G5	0xB1CA
NEC16	Daewoo	VCR	0x0015
KASEIKYO	Technics EUR646497	SA-AX 730 AV receiver	0x2002
	Panasonic TV	TX-L32EW6 TV	0x2002
RC5	Loewe Assist/RC3/RC4	TV (remote control in TV mode)	0x0000
RC6	Philips TV	TV (remote control in TV mode)	0x0000
SIRCS	Sony RM-816	TV (remote control in TV mode)	0x0000
DENON	DENON RC970	AVR3805 AV receiver	0x0008
	DENON RC970	DVD/CD player	0x0002
	DENON RC970	Tuner	0x0006
SAMSUNG32	Samsung AA59-00484A	LE40D550 TV	0x0707
	LG AKB72033901	BD370 Blu-Ray player	0x2D2D
APPLE	Apple	Apple iPod Dock	0x0020

Cameras

IRMP supports more and more camera remotes like:

- PENTAX
- Nikon

The array of commands is quite limited. Cameras understand only the shutter release command.



PENTAX protocol

Here is a short table for PENTAX cameras:

Command	Function
0x0000	Shutter release
0x0001	change zoom level

Because there is no address designated in the PENTAX protocol, for transmitting, it should be set to 0x0000 in IRSND.

For Nikon cameras, a crystal should be used because these cameras are quite sensitive to timing accuracy.

IR Keyboards

From version 1.7.0 on, IRMP also supports IR keyboards, namely the FDC-3402 ~~from www.pollin.de (partno. 711 056) for less than 2 Euro.~~ (not available as of 19.09.2017)


FDC-3402 keyboard

On detection of a key press the following data is returned:

```
Protocol number
(irmp_data.protocol): 18
Address
(irmp_data.address) : 0x003F
```

The following values are returned as commands (irmp_data.command) :

Code	Key	Code	Key	Code	Key	Code	Key	Code	Key	Code	Key	Code	Key	Code	Key
0x0000		0x0010	TAB	0x0020	's'	0x0030	'c'	0x0040		0x0050	HOME	0x0060		0x0070	MENUE
0x0001	'^'	0x0011	'q'	0x0021	'd'	0x0031	'v'	0x0041		0x0051	END	0x0061		0x0071	BACK
0x0002	'1'	0x0012	'w'	0x0022	'f'	0x0032	'b'	0x0042		0x0052		0x0062		0x0072	FORWARD
0x0003	'2'	0x0013	'e'	0x0023	'g'	0x0033	'n'	0x0043		0x0053	UP	0x0063		0x0073	ADDRESS
0x0004	'3'	0x0014	'r'	0x0024	'h'	0x0034	'm'	0x0044		0x0054	DOWN	0x0064		0x0074	WINDOW
0x0005	'4'	0x0015	't'	0x0025	'j'	0x0035	','	0x0045		0x0055	PAGE_UP	0x0065		0x0075	1ST_PAGE
0x0006	'5'	0x0016	'z'	0x0026	'k'	0x0036	'.'	0x0046		0x0056	PAGE_DOWN	0x0066		0x0076	STOP
0x0007	'6'	0x0017	'u'	0x0027	'l'	0x0037	'-'	0x0047		0x0057		0x0067		0x0077	MAIL
0x0008	'7'	0x0018	'i'	0x0028	'ö'	0x0038		0x0048		0x0058		0x0068		0x0078	FAVORITES
0x0009	'8'	0x0019	'o'	0x0029	'ä'	0x0039	SHIFT_RIGHT	0x0049		0x0059	RIGHT	0x0069		0x0079	NEW_PAGE
0x000A	'9'	0x001A	'p'	0x002A	'#'	0x003A	CTRL	0x004A		0x005A		0x006A		0x007A	SETUP
0x000B	'0'	0x001B	'ü'	0x002B	CR	0x003B		0x004B	INSERT	0x005B		0x006B		0x007B	FONT
0x000C	'ß'	0x001C	'+'	0x002C	SHIFT_LEFT	0x003C	ALT_LEFT	0x004C	DELETE	0x005C		0x006C		0x007C	PRINT
0x000D	'´'	0x001D		0x002D	'<'	0x003D	SPACE	0x004D		0x005D		0x006D		0x007D	
0x000E		0x001E	CAPSLOCK	0x002E	'y'	0x003E	ALT_RIGHT	0x004E		0x005E		0x006E	ESCAPE	0x007E	ON_OFF

0x000F	BACKSPACE	0x001F	'a'	0x002F	'x'	0x003F		0x004F	LEFT	0x005F		0x006F		0x007F	

Special keys on the left:

Code	Key
0x0400	KEY_MOUSE_1
0x0800	KEY_MOUSE_2

The above values are for pressing a key. On release, IRMP sets also bit 8 (0x80) in the command.

Example:

```
Key 'a' pressed:   0x001F
Key 'a' released:  0x009F
```

The ON/OFF key is an exception: This key only sends a code for a key press, not for the release.

If a key is held, this is indicated in irmp_data.flag.

Example:

```
                     command   flag
Key 'a' pressed:   0x001F    0x00
Key 'a' pressed:   0x001F    0x01
Key 'a' pressed:   0x001F    0x01
Key 'a' pressed:   0x001F    0x01
....
Key 'a' released:  0x009F    0x00
```

When key combinations (like a capital 'A') are pressed, then the return values are a sequence like this:

```
Left SHIFT-key pressed:    0x0002
Key 'a' pressed:           0x001F
Key 'a' released:          0x009F
Left SHIFT-key released:   0x0082
```

In irmp.c you will find a function get_fdc_key() for the Linux version, which can be used as a template to convert the FDC keycodes into the corresponding ASCII codes. This function can be used either locally on the MCU to decode the keycodes, or on the host system (e.g. PC) where the IRMP data structure is sent to. Therefore the function including preprocessor constants should be copied to your application code.

Here is an excerpt:

```
#define STATE_LEFT_SHIFT    0x01
#define STATE_RIGHT_SHIFT   0x02
#define STATE_LEFT_CTRL     0x04
#define STATE_LEFT_ALT      0x08
#define STATE_RIGHT_ALT     0x10
```

```
#define KEY_ESCAPE          0x1B            // keycode = 0x006e
#define KEY_MENUE           0x80            // keycode = 0x0070
#define KEY_BACK            0x81            // keycode = 0x0071
#define KEY_FORWARD         0x82            // keycode = 0x0072
#define KEY_ADDRESS         0x83            // keycode = 0x0073
#define KEY_WINDOW          0x84            // keycode = 0x0074
#define KEY_1ST_PAGE        0x85            // keycode = 0x0075
#define KEY_STOP            0x86            // keycode = 0x0076
#define KEY_MAIL            0x87            // keycode = 0x0077
#define KEY_FAVORITES       0x88            // keycode = 0x0078
#define KEY_NEW_PAGE        0x89            // keycode = 0x0079
#define KEY_SETUP           0x8A            // keycode = 0x007a
#define KEY_FONT            0x8B            // keycode = 0x007b
#define KEY_PRINT           0x8C            // keycode = 0x007c
#define KEY_ON_OFF          0x8E            // keycode = 0x007c

#define KEY_INSERT          0x90            // keycode = 0x004b
#define KEY_DELETE          0x91            // keycode = 0x004c
#define KEY_LEFT            0x92            // keycode = 0x004f
#define KEY_HOME            0x93            // keycode = 0x0050
#define KEY_END             0x94            // keycode = 0x0051
#define KEY_UP              0x95            // keycode = 0x0053
#define KEY_DOWN            0x96            // keycode = 0x0054
#define KEY_PAGE_UP         0x97            // keycode = 0x0055
#define KEY_PAGE_DOWN       0x98            // keycode = 0x0056
#define KEY_RIGHT           0x99            // keycode = 0x0059
#define KEY_MOUSE_1         0x9E            // keycode = 0x0400
#define KEY_MOUSE_2         0x9F            // keycode = 0x0800

static uint8_t
get_fdc_key (uint16_t cmd)
{
    static uint8_t key_table[128] =
    {
     // 0    1    2    3    4    5    6    7    8    9    A    B    C    D    E
F
        0,   '^', '1', '2', '3', '4', '5', '6', '7',  '8',  '9',  '0',  0xDF, '´',
0,   '\b',
        '\t', 'q', 'w', 'e', 'r', 't', 'z', 'u', 'i',  'o',  'p',  0xFC, '+',  0,
0,   'a',
        's',  'd', 'f', 'g', 'h', 'j', 'k', 'l', 0xF6, 0xE4, '#',  '\r', 0,    '<',
'y', 'x',
        'c',  'v', 'b', 'n', 'm', ',', '.', '-', 0,    0,    0,    0,    0,    ' ',
0,   0,

        0,   '°', '!', '"', '§', '$', '%', '&', '/',  '(',  ')',  '=',  '?',  '`',
0,   '\b',
        '\t', 'Q', 'W', 'E', 'R', 'T', 'Z', 'U', 'I',  'O',  'P',  0xDC, '*',  0,
0,   'A',
        'S',  'D', 'F', 'G', 'H', 'J', 'K', 'L', 0xD6, 0xC4, '\'', '\r', 0,    '>',
'Y', 'X',
        'C',  'V', 'B', 'N', 'M', ';', ':', '_', 0,    0,    0,    0,    0,    ' ',
0,   0
    };
    static uint8_t state;

    uint8_t key = 0;

    switch (cmd)
    {
        case 0x002C: state |=  STATE_LEFT_SHIFT;    break;              // pressed
left shift
        case 0x00AC: state &= ~STATE_LEFT_SHIFT;    break;              // released
```

```
    left shift
            case 0x0039: state |=  STATE_RIGHT_SHIFT;   break;              // pressed
    right shift
            case 0x00B9: state &= ~STATE_RIGHT_SHIFT;   break;              // released
    right shift
            case 0x003A: state |=  STATE_LEFT_CTRL;     break;              // pressed
    left ctrl
            case 0x00BA: state &= ~STATE_LEFT_CTRL;     break;              // released
    left ctrl
            case 0x003C: state |=  STATE_LEFT_ALT;      break;              // pressed
    left alt
            case 0x00BC: state &= ~STATE_LEFT_ALT;      break;              // released
    left alt
            case 0x003E: state |=  STATE_RIGHT_ALT;     break;              // pressed
    left alt
            case 0x00BE: state &= ~STATE_RIGHT_ALT;     break;              // released
    left alt

            case 0x006e: key = KEY_ESCAPE;              break;
            case 0x004b: key = KEY_INSERT;              break;
            case 0x004c: key = KEY_DELETE;              break;
            case 0x004f: key = KEY_LEFT;                break;
            case 0x0050: key = KEY_HOME;                break;
            case 0x0051: key = KEY_END;                 break;
            case 0x0053: key = KEY_UP;                  break;
            case 0x0054: key = KEY_DOWN;                break;
            case 0x0055: key = KEY_PAGE_UP;             break;
            case 0x0056: key = KEY_PAGE_DOWN;           break;
            case 0x0059: key = KEY_RIGHT;               break;
            case 0x0400: key = KEY_MOUSE_1;             break;
            case 0x0800: key = KEY_MOUSE_2;             break;

            default:
            {
                if (!(cmd & 0x80))                      // pressed key
                {
                    if (cmd >= 0x70 && cmd <= 0x7F)     // function keys
                    {
                        key = cmd + 0x10;               // 7x -> 8x
                    }
                    else if (cmd < 64)                  // key listed in key_table
                    {
                        if (state & (STATE_LEFT_ALT | STATE_RIGHT_ALT))
                        {
                            switch (cmd)
                            {
                                case 0x0003: key = 0xB2;    break; // ²
                                case 0x0008: key = '{';     break;
                                case 0x0009: key = '[';     break;
                                case 0x000A: key = ']';     break;
                                case 0x000B: key = '}';     break;
                                case 0x000C: key = '\\';    break;
                                case 0x001C: key = '~';     break;
                                case 0x002D: key = '|';     break;
                                case 0x0034: key = 0xB5;    break; // µ
                            }
                        }
                        else if (state & (STATE_LEFT_CTRL))
                        {
                            if (key_table[cmd] >= 'a' && key_table[cmd] <= 'z')
                            {
                                key = key_table[cmd] - 'a' + 1;
                            }
```

```
                        else
                        {
                            key = key_table[cmd];
                        }
                    }
                    else
                    {
                        int idx = cmd + ((state & (STATE_LEFT_SHIFT |
STATE_RIGHT_SHIFT)) ? 64 : 0);

                        if (key_table[idx])
                        {
                            key = key_table[idx];
                        }
                    }
                }
            }
            break;
        }
    }

    return (key);
}
```

As a final example, use of the get_fdc_key() function:

```
    if (irmp_get_data (&irmp_data))
    {
        uint8_t key;

        if (irmp_data.protocol == IRMP_FDC_PROTOCOL &&
            (key = get_fdc_key (irmp_data.command)) != 0)
        {
            if ((key >= 0x20 && key < 0x7F) || key >= 0xA0) // show only printable
characters
            {
                printf ("ascii-code = 0x%02x, character = '%c'\n", key, key);
            }
            else // it's a non-printable key
            {
                printf ("ascii-code = 0x%02x\n", key);
            }
        }
    }
```

Non-printable characters are coded as follows:

Key	Constant	Value
ESC	KEY_ESCAPE	0x1B
Menu	KEY_MENUE	0x80
Back	KEY_BACK	0x81
Forward	KEY_FORWARD	0x82
Adress	KEY_ADDRESS	0x83
Window	KEY_WINDOW	0x84

1. Page	KEY_1ST_PAGE	0x85
Stop	KEY_STOP	0x86
Mail	KEY_MAIL	0x87
Fav.	KEY_FAVORITES	0x88
New Page	KEY_NEW_PAGE	0x89
Setup	KEY_SETUP	0x8A
Font	KEY_FONT	0x8B
Print	KEY_PRINT	0x8C
On/Off	KEY_ON_OFF	0x8E
Backspace	'\b'	0x08
CR/ENTER	'\r'	0x0C
TAB	'\t'	0x09
Insert	KEY_INSERT	0x90
Delete	KEY_DELETE	0x91
Cursor left	KEY_LEFT	0x92
Pos1	KEY_HOME	0x93
End	KEY_END	0x94
Cursor right	KEY_UP	0x95
Cursor down	KEY_DOWN	0x96
Page up	KEY_PAGE_UP	0x97
Page down	KEY_PAGE_DOWN	0x98
Cursor left	KEY_RIGHT	0x99
Left Mousebutton	KEY_MOUSE_1	0x9E
Right Mousebutton	KEY_MOUSE_2	0x9F

The get_fdc_key() function considers the state of the Shift, Ctrl, and Alt keys. As a result, not only capital letters can be entered, but also special characters with Alt + key combinations, e.g. Alt + m -> µ or Alt + q -> @. You can also send Ctrl + A to Ctrl + Z by using the Ctrl key. The Caps Lock key is ignored, as I regard this key as the most unnecessary key at all ;-)

Appendix

IR Protocols in Detail

Pulse Distance Protocols

Pulse Distance Coding

NEC + extended NEC

NEC + extended NEC	Value
Frequency	36 kHz / 38 kHz
Coding	pulse distance
Frame	1 start bit + 32 data bits + 1 stop bit
Data NEC	8 address bits + 8 inverted address bits + 8 command bits + 8 inverted command bits
Data ext. NEC	16 address bits + 8 command bits + 8 inverted command bits
Start bit	9000µs pulse, 4500µs pause
0 bit	560µs pulse, 560µs pause
1 bit	560µs pulse, 1690µs pause
Stop bit	560µs pulse
Repetition	none
Key repeat	9000µs pulse, 2250µs pause, 560µs pulse, ~100ms pause
Bit order	LSB first

JVC

JVC	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 16 data bits + 1 stop bit
Data	4 address bits + 12 command bits
Start bit	9000µs pulse, 4500µs pause, 6000µs pause if key repeat
0 bit	560µs pulse, 560µs pause
1 bit	560µs pulse, 1690µs pause
Stop bit	560µs pulse
Repetition	none
Key repeat	after pause of 25ms
Bit order	LSB first

NEC16

NEC16	Value
Frequency	38 kHz

Coding	pulse distance
Frame	1 start bit + 8 address bits + 1 sync bit + 8 data bits + 1 stop bit
Start bit	9000µs pulse, 4500µs pause
sync bit	560µs pulse, 4500µs pause
0 bit	560µs pulse, 560µs pause
1 bit	560µs pulse, 1690µs pause
Stop bit	560µs pulse
Repetition	none
Key repeat	after pause of 25ms?
Bit order	LSB first

NEC42

NEC42	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 42 data bits + 1 stop bit
Data	13 address bits + 13 inverted address bits + 8 command bits + 8 inverted command bits
Start bit	9000µs pulse, 4500µs pause
0 bit	560µs pulse, 560µs pause
1 bit	560µs pulse, 1690µs pause
Stop bit	560µs pulse
Repetition	none
Key repeat	after 110ms (beginning from start bit), 9000µs pulse, 2250µs pause, 560µs pulse
Bit order	LSB first

ACP24

ACP24	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 70 data bits + 1 stop bit
Data	0 address bits + 70 command bits
Start bit	390µs pulse, 950µs pause
0 bit	390µs pulse, 950µs pause
1 bit	390µs pulse, 13000µs pause
Stop bit	390µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first

LGAIR

LGAIR	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 28 data bits + 1 stop bit
Data	8 address bits + 16 command bits + 4 checksum bits
Start bit	9000µs pulse, 4500µs pause (identical to NEC)
0 bit	560µs pulse, 560µs pause (identical to NEC)
1 bit	560µs pulse, 1690µs pause (identical to NEC)
Stop bit	560µs pulse (identical to NEC)
Repetition	none
Key repeat	unknown
Bit order	MSB first (**differs** from NEC)

SAMSUNG

SAMSUNG	Value
Frequency	?? kHz
Coding	pulse distance
Frame	1 start bit + 16 data(1) bits + 1 sync bit + 20 data(2)-bits + 1 stop bit
Data(1)	16 address bits
Data(2)	4 ID bits + 8 command bits + 8 inverted command bits
Start bit	4500µs pulse, 4500µs pause
0 bit	550µs pulse, 550µs pause
1 bit	550µs pulse, 1650µs pause
sync bit	550µs pulse, 4500µs pause
Stop bit	550µs pulse
Repetition	none
Key repeat	after approx. 100ms
Bit order	LSB first

SAMSUNG32

SAMSUNG32	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 32 data bits + 1 stop bit
Data	16 address bits + 16 command bits
Start bit	4500µs pulse, 4500µs pause
0 bit	550µs pulse, 550µs pause
1 bit	550µs pulse, 1650µs pause
Stop bit	550µs pulse

Repetition	none
Key repeat	after approx. 47msec
Bit order	LSB first

SAMSUNG48

SAMSUNG48	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 48 data bits + 1 stop bit
Data	16 address bits + 32 command bits
Command	8 bits + 8 inverted bits + 8 bits + 8 inverted bits
Start bit	4500µs pulse, 4500µs pause
0 bit	550µs pulse, 550µs pause
1 bit	550µs pulse, 1650µs pause
Stop bit	550µs pulse
Repetition	one after approx. 5 msec
Key repeat	after approx. 45 msec
Bit order	LSB first

MATSUSHITA

MATSUSHITA	Value
Frequency	36 kHz
Coding	pulse distance, timing identical to TECHNICS
Frame	1 start bit + 24 data bits + 1 stop bit
Data	6 customer bits + 6 command bits + 12 address bits
Start bit	3488µs pulse, 3488µs pause
0 bit	872µs pulse, 872µs pause
1 bit	872µs pulse, 2616µs pause
Stop bit	872µs pulse
Repetition	none
Key repeat	after 40ms pause
Bit order	LSB first?

TECHNICS

TECHNICS	Value
Frequency	36 kHz?
Coding	pulse distance, timing identical to MATSUSHITA
Frame	1 start bit + 22 data bits + 1 stop bit
Data	11 command bits + 11 inverted command bits
Start bit	3488µs pulse, 3488µs pause

0 bit	872µs pulse, 872µs pause
1 bit	872µs pulse, 2616µs pause
Stop bit	872µs pulse
Repetition	none
Key repeat	after 40ms pause
Bit order	LSB first?

KASEIKYO

KASEIKYO	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 48 data bits + 1 stop bit
Data	16 customer bits + 4 parity bits + 4 genre1 bits + 4 genre2 bits + 10 command bits + 2 ID bits + 8 parity bits
Start bit	3380µs pulse, 1690µs pause
0 bit	423µs pulse, 423µs pause
1 bit	423µs pulse, 1269µs pause
Stop bit	423µs pulse
Repetition	none
Key repeat	after approx. 80ms pause
Bit order	LSB first?

RECS80

RECS80	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bits + 10 data bits + 1 stop bit
Data	1 toggle bit + 3 address bits + 6 command bits
Start bit	158µs pulse, 7432µs pause
0 bit	158µs pulse, 4902µs pause
1 bit	158µs pulse, 7432µs pause
Stop bit	158µs pulse
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

RECS80EXT

RECS80EXT	Value
Frequency	38 kHz
Coding	pulse distance
Frame	2 start bits + 11 data bits + 1 stop bit

Data	1 toggle bit + 4 address bits + 6 command bits
Start bit	158µs pulse, 3637µs pause
0 bit	158µs pulse, 4902µs pause
1 bit	158µs pulse, 7432µs pause
Stop bit	158µs pulse
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

DENON

DENON	Value
Frequency	38 kHz (in practice, theoretically: 32 kHz)
Coding	pulse distance
Frame	0 start bits + 15 data bits + 1 stop bit
Data	5 address bits + 10 command bits
Command	6 data bits + 2 extension bits + 2 data construction bits (normal: 00, inverted: 11)
Start bit	none
0 bit	310µs pulse, 745µs pause (in practice, theoretically: 275µs pulse, 775µs pause)
1 bit	310µs pulse, 1780µs pause (in practice, theoretically: 275µs pulse, 1900µs pause)
Stop bit	310µs pulse (310µs pulse, 745µs pause (in practice, theoretically: 275µs pulse)
Repetition	after 65ms with inverted command bits (data construction bits = 11)
Key repeat	both frames after 65ms
Bit order	MSB first

APPLE

APPLE	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 32 data bits + 1 stop bit
Data	16 address bits + 11100000 + 8 command bits
Start bit	see NEC
0 bit	see NEC
1 bit	see NEC
Stop bit	see NEC
Repetition	none
Key repeat	after approx. 100ms
Bit order	LSB first

BOSE

BOSE	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 16 data bits + 1 stop bit
Data	0 address bits + 8 command bits + 8 inverted command bits
Start bit	1060µs pulse, 1425µs pause
0 bit	550µs pulse, 437µs pause
1 bit	550µs pulse, 1425µs pause
Stop bit	550µs pulse
Repetition	none
Key repeat	unknown
Bit order	LSB first

B&O

B&O	Value
Frequency	455 kHz
Coding	pulse distance
Frame	4 start bits + 16 data bits + 1 trailer bit + 1 stop bit
Data	0 address bits + 16 command bits
Start bit 1	200µs pulse, 2925µs pause
Start bit 2	200µs pulse, 2925µs pause
Start bit 3	200µs pulse, 15425µs pause
Start bit 4	200µs pulse, 2925µs pause
0 bit	200µs pulse, 2925µs pause
1 bit	200µs pulse, 9175µs pause
R bit	200µs pulse, 6050µs pause, repeats the last bit
Trailer bit	200µs pulse, 12300µs pause
Stop bit	200µs pulse
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

FDC

FDC	Value
Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 40 data bits + 1 stop bit
Data	8 address bits + 12 x 0 bits + 4 press/release bits + 8 command bits + 8 inverted command bits

Start bit	2085µs pulse, 966µs pause
0 bit	300µs pulse, 220µs pause
1 bit	300µs pulse, 715µs pause
Stop bit	300µs pulse
Repetition	none
Press Key	press/release bits = 0000
Release Key	press/release bits = 1111
Key repeat	after pause of approx. 60ms
Bit order	LSB first

Nikon

Nikon	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 2 data bits + 1 stop bit
Data	2 command bits
Start bit	2200µs pulse, 27100µs pause
0 bit	500µs pulse, 1500µs pause
1 bit	500µs pulse, 3500µs pause
Stop bit	500µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first

PANASONIC

PANASONIC	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 56 data bits + 1 stop bit
Data	24 bits (010000000000010000000001) + 16 address bits + 16 command bits
Start bit	3600µs pulse, 1600µs pause
0 bit	565µs pulse, 316µs pause
1 bit	565µs pulse, 1140µs pause
Stop bit	565µs pulse
Repetition	none
Key repeat	unknown
Bit order	LSB first?

PENTAX

PENTAX	Value

Frequency	38 kHz
Coding	pulse distance
Frame	1 start bit + 6 data bits + 1 stop bit
Data	6 command bits
Start bit	2200µs pulse, 27100µs pause
0 bit	1000µs pulse, 1000µs pause
1 bit	1000µs pulse, 3000µs pause
Stop bit	1000µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first

KATHREIN

KATHREIN	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 11 data bits + 1 stop bit
Data	4 address bits + 7 command bits
Start bit	210µs pulse, 6218µs pause
0 bit	210µs pulse, 1400µs pause
1 bit	210µs pulse, 3000µs pause
Stop bit	210µs pulse
Repetition	none
Key repeat	after 35ms?
Bit order	MSB first

LEGO

LEGO	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 16 data bits + 1 stop bit
Data	12 command bits + 4 crc bits
Start bit	158µs pulse, 1026µs pause
0 bit	158µs pulse, 263µs pause
1 bit	158µs pulse, 553µs pause
Stop bit	158µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first

VINCENT

VINCENT	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 32 data bits + 1 stop bit
Data	16 address bits + 8 command bits + 8 repeated command bits
Start bit	2500µs pulse, 4600µs pause
0 bit	550µs pulse, 550µs pause
1 bit	550µs pulse, 1540µs pause
Stop bit	550µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first?

THOMSON

THOMSON	Value
Frequency	33 kHz
Coding	pulse distance
Frame	0 start bits + 12 data bits + 1 stop bit
Data	4 address bits + 1 toggle bit + 7 command bits
0 bit	550µs pulse, 2000µs pause
1 bit	550µs pulse, 4500µs pause
Stop bit	550µs pulse
Repetition	none
Key repeat	after 35ms
Bit order	MSB first?

TELEFUNKEN

TELEFUNKEN	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 15 data bits + 1 stop bit
Data	0 address bits + 15 command bits
Start bit	600µs pulse, 1500µs pause
0 bit	600µs pulse, 600µs pause
1 bit	600µs pulse, 1500µs pause
Stop bit	600µs pulse
Repetition	none
Key repeat	unknown
Bit order	MSB first?

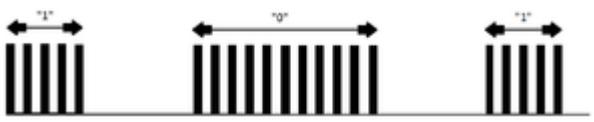
RCCAR

RCCAR	Value
Frequency	38 kHz?
Coding	pulse distance
Frame	1 start bit + 13 data bits + 1 stop bit
Data	13 command bits
Start bit	2000µs pulse, 2000µs pause
0 bit	600µs pulse, 900µs pause
1 bit	600µs pulse, 450µs pause
Stop bit	600µs pulse
Repetition	none
Key repeat	after 40ms?
Bit order	LSB first

RCMM

RCMM	Value
Frequency	36 kHz
Coding	pulse distance
Frame RCMM32	1 start bit + 32 data bits + 1 stop bit
Frame RCMM24	1 start bit + 24 data bits + 1 stop bit
Frame RCMM12	1 start bit + 12 data bits + 1 stop bit
Data RCMM32	16 address bits (= 4 mode bits + 12 device bits) + 1 toggle bit + 15 command bits
Data RCMM24	16 address bits (= 4 mode bits + 12 device bits) + 1 toggle bit + 7 command bits
Data RCMM12	4 address bits (= 2 mode bits + 2 device bits) + 8 command bits
Start bit	500µs pulse, 220µs pause
00 bits	230µs pulse, 220µs pause
01 bits	230µs pulse, 380µs pause
10 bits	230µs pulse, 550µs pause
11 bits	230µs pulse, 720µs pause
Stop bit	230µs pulse
Repetition	none
Key repeat	after 80ms
Bit order	LSB first

Pulse Width Protocols

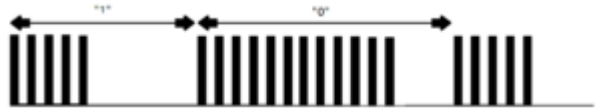


Pulse Width Coding

SIRCS

SIRCS	Value
Frequency	40 kHz
Coding	pulse width
Frame	1 start bit + 12-20 data bits, no stop bit
Data	7 command bits + 5 address bits + up to 8 additional bits
Start bit	2400µs pulse, 600µs pause
0 bit	600µs pulse, 600µs pause
1 bit	1200µs pulse, 600µs pause
Repetition	twice after approx. 25ms, that means: 2nd and 3rd frame
Key repeat	starting with 4th identical frame, distance approx. 25ms
Bit order	LSB first

Pulse distance Width Protocols



Pulse Distance Width Coding

NUBERT

NUBERT	Value
Frequency	36 kHz?
Coding	pulse distance width
Frame	1 start bit + 10 data bits + 1 stop bit
Data	0 address bits + 10 command bits ?
Start bit	1340µs pulse, 340µs pause
0 bit	500µs pulse, 1300µs pause
1 bit	1340µs pulse, 340µs pause
Stop bit	500µs pulse
Repetition	once after 35ms
Key repeat	3rd, 5th, 7th etc. indentical frame
Bit order	MSB first?

FAN

This protocol is very similar to NUBERT, but here it will be sent only one frame. Additionally there are 11 instead of 10 data bits and no stop bit. The pause time between frame repetitions is substantial lower.

FAN	Value
Frequency	36 kHz
Coding	pulse distance width

Frame	1 start bit + 11 data bits + 0 stop bits
Data	0 address bits + 11 command bits
Start bit	1280µs pulse, 380µs pause
0 bit	380µs pulse, 1280µs pause
1 bit	1280µs pulse, 380µs pause
Stop bit	500µs pulse
Repetition	none
Key repeat	after 6.6ms pause
Bit order	MSB first

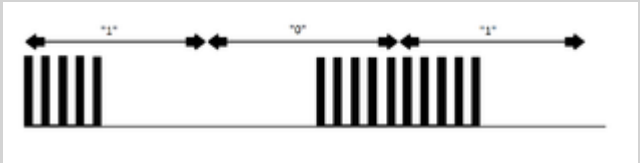
SPEAKER

SPEAKER	Value
Frequency	38 kHz?
Coding	pulse distance width
Frame	1 start bit + 10 data bits + 1 stop bit
Data	0 address bits + 10 command bits ?
Start bit	440µs pulse, 1250µs pause
0 bit	440µs pulse, 1250µs pause
1 bit	1250µs pulse, 440µs pause
Stop bit	440µs pulse
Repetition	once after approx. 38ms
Key repeat	3rd, 5th, 7th ... identical frame
Bit order	MSB first?

ROOMBA

ROOMBA	Value
Frequency	38 kHz?
Coding	pulse distance width
Frame	1 start bit + 7 data bits + 0 stop bits
Data	0 address bits + 7 command bits
Start bit	2790µs pulse, 930µs pause
0 bit	930µs pulse, 2790µs pause
1 bit	2790µs pulse, 930µs pause
Stop bit	no stop bit
Repetition	3 times after 18ms?
Key repeat	unknown
Bit order	MSB first

Biphase Protocols



Biphase Coding

RC5 + RC5X

RC5 + RC5X	Value
Frequency	36 kHz
Coding	Biphase (Manchester)
Frame RC5	2 start bits + 12 data bits + 0 stop bits
Data RC5	1 toggle bit + 5 address bits + 6 command bits
Frame RC5X	1 start bit + 13 data bits + 0 stop bits
Data RC5X	1 inverteds command bit + 1 toggle bit + 5 address bits + 6 command bits
Start bit	889µs pause, 889µs pulse
0 bit	889µs pulse, 889µs pause
1 bit	889µs pause, 889µs pulse
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

RCII

RCII	Value
Frequency	31.25 kHz
Coding	Biphase (Manchester)
Frame	1 pre bit + 1 start bit + 9 data bits + 0 stop bits
Data	0 address bits + 9 command bits
Pre Bit	512µs pulse, 2560µs pause
Start bit	1024µs pulse, **no space**
0 bit	512µs pause, 512µs pulse
1 bit	512µs pulse, 512µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 118ms
Remarks	An end command (111111111 = 0x1FF) is sent immediately after the button is released.
Bit-Order	MSB first

S100

Similar to RC5x, but 14 instead of 13 data bits and 56kHz modulation

S100	Value
Frequency	56 kHz

Coding	Biphase (Manchester)
Frame	1 start bit + 14 data bits + 0 stop bits
Data	1 inverted command bit + 1 toggle bit + 5 address bits + 7 command bits
Start bit	889µs pause, 889µs pulse
0 bit	889µs pulse, 889µs pause
1 bit	889µs pause, 889µs pulse
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

RC6 + RC6A

RC6 + RC6A	Value
Frequency	36 kHz
Coding	Biphase (Manchester)
Frame RC6	1 start bit + 1 bit "1" + 3 mode bits ("000") + 1 toggle bit + 16 data bits + 2666µs pause
Frame RC6A	1 start bit + 1 bit "1" + 3 mode bits ("110") + 1 toggle bit + 31 data bits + 2666µs pause
Data RC6	8 address bits + 8 Command Bits
Data RC6A	"1" + 14 customer bits + 8 system bits + 8 command bits
Data RC6A Pace (Sky)	"1" + 3 mode bits ("110") + 1 toggle bit (UNUSED "0") + 16 bits + 1 toggle(!) + 15 command bits
Start bit	2666µs pulse, 889µs pause
Toggle 0 bit	889µs pause, 889µs pulse
Toggle 1 bit	889µs pulse, 889µs pause
0 bit	444µs pause, 444µs pulse
1 bit	444µs pulse, 444µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 100ms
Bit order	MSB first

GRUNDIG + NOKIA

GRUNDIG + NOKIA	Value
Frequency	38 kHz (?)
Coding	Biphase (Manchester)
Frame packet	1 start frame + 19.968ms pause + N data frames + 117.76ms pause + 1 stop frame
Start frame	1 pre bit + 1 start bit + 9 data bits (all 1) + 0 stop bits
Data frame	1 pre bit + 1 start bit + 9 data bits + 0 stop bits

Stop frame	1 pre bit + 1 start bit + 9 data bits (all 1) + 0 stop bits
Data Grundig	9 command bits + 0 address bits
Data Nokia	8 command bits + 8 address bits
Pre bit	528µs pulse, 2639µs pause
Start bit	528µs pulse, 528µs pause
0 bit	528µs pause, 528µs pulse
1 bit	528µs pulse, 528µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 117.76ms
Bit order	LSB first

IR60 (SDA2008)

IR60	Value
Frequency	30 kHz
Coding	Biphase (Manchester)
Start frame	1 start bit + 101111 + 0 stop bits + 22ms pause
Data frame	1 start bit + 7 data bits + 0 stop bits
Data	0 address bits + 7 command bits
Start bit	528µs pulse, 2639µs pause
0 bit	528µs pause, 528µs pulse
1 bit	528µs pulse, 528µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	after approx. 117.76ms
Bit order	LSB first

SIEMENS + RUWIDO

SIEMENS + RUWIDO	Value
Frequency	36 kHz? (Merlin keyboard with Ruwido protocol: 56 kHz)
Coding	Biphase (Manchester)
Frame Siemens	1 start bit + 22 data bits + 0 stop bits
Frame Ruwido	1 start bit + 17 data bits + 0 stop bits
Data Siemens	11 address bits + 10 command bits + 1 inverted bit (preceding bit inverted)
Data Ruwido	9 address bits + 7 command bits + 1 inverted bit (preceding bit inverted)
Start bit	275µs pulse, 275µs pause
0 bit	275µs pause, 275µs pulse
1 bit	275µs pulse, 275µs pause
Stop bit	no stop bit
Repetition	once with repeat bit set (?)

Key repeat	after approx. 100ms (?)
Bit order	MSB first

A1TVBOX

A1TVBOX	Value
Frequency	38 kHz?
Coding	Biphase (Manchester) asymmetric
Frame	2 start bits + 16 data bits + 0 stop bits
Data	8 address bits + 8 command bits
Start bits	"10", also 250µs pulse, 150µs + 150µs pause, 250µs pulse
0 bit	150µs pause, 250µs pulse
1 bit	250µs pulse, 150µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	unknown
Bit order	MSB first

MERLIN

MERLIN	Value
Frequency	56 kHz
Coding	Biphase (Manchester) asymmetric
Frame	2 start bits + 18 data bits + 0 stop bits
Data	8 address bits + 10 command bits
Start bits	"10", also 210µs pulse, 210µs + 210µs pause, 210µs pulse
0 bit	210µs pause, 210µs pulse
1 bit	210µs pulse, 210µs pause
Stop bit	no stop bit
Repetition	none
Key repeat	unknown
Bit order	MSB first

ORTEK

ORTEK	Value
Frequency	38 kHz?
Coding	Biphase (Manchester) symmetric
Frame	2 start bits + 18 data bits + 0 stop bits
Data	6 address bits + 2 special bits + 6 command bits + 4 special bits
Start bit	2000µs pulse, 1000µs pause
0 bit	500µs pause, 500µs pulse
1 bit	500µs pulse, 500µs pause

Stop bit	no stop bit
Repetition	2 additional frames with special bits set
Key repeat	only repeats the 2nd frame
Bit order	MSB first

Pulse Position Protocols

NETBOX

NETBOX	Value
Frequency	38 kHz?
Coding	pulse position
Frame	1 start bit + 16 data bits, no stop bit
Data	3 address bits + 13 command bits
Start bit	2400µs pulse, 800µs pause
Bit Length	800µs
Repetition	none
Key repeat	after approx. 35ms?
Bit order	LSB first

Software History

Changes of IRMP in 3.2.x

Version 3.2.6:

- 2021-01-27: **New IR Protocol:** MELINERA
- 2021-01-27: Protocol LEGO: Improved timing
- 2021-01-27: Protocol RUWIDO: Improved timing
- 2021-01-27: Protocol NEC: Implemented send of raw NEC repetition frames

Version 3.2.3:

- 2020-08-15: **New RF Protocol:** RF_MEDION

Version 3.2.2:

- 2020-07-09: Additional recognition of the radio channels with the RF_X10 protocol
- 2020-07-09: Improved RF frame detection with new stop bit handling.
- 2020-07-09: Improved detection of RF_GEN24 protocols
- 2020-07-09: **NEU:** Detection if/when a remote control button is released, see chapter **Debouncing**.

Version 3.2.1:

- 2020-06-22: Mini bugfix

Version 3.2.0:

- 2020-06-22: Support of 433MHz RF modules
- 2020-06-22: **New protocol**: RF_GEN24
- 2020-06-22: **New protocol**: X10

Older Versions

- 2019-08-26: **New protocol**: METZ
- 2019-08-26: **New protocol**: ONKYO
- 2018-09-10: **New protocol**: RCII
- 2018-09-06: Added support of STM32 mit HAL-Library
- 2018-08-30: New option: IRMP_USE_IDLE_CALL
- 2018-08-29: Port to ChibiOS
- 2018-08-29: New protocol: GREE
- 2018-02-19: corrected handling of irmp_flags for invalid frames
- 2017-08-25: New protocol: IRMP16 for transparent 16 bit data communication
- 2016-11-18: Corrected buffer overflow in irmp-main-avr-uart.c
- 2016-09-19: New protocol VINCENT
- 2016-09-09: New protocol Mitsubishi Heavy (air conditioner)
- 2016-09-09: Some modifications for Compiler PIC C18
- 2016-01-16: Some corrections of port to ESP8266
- 2016-01-16: Added port to MBED
- 2016-01-16: Added several hardware dependent example main source files
- 2015-11-17: **New protocol**: PANASONIC (Beamer)
- 2015-11-17: Port to ESP8266
- 2015-11-17: Port to Teensy (3.x)
- 2015-11-10: Added support for STM8 microcontroller
- 2015-09-20: **New protocol**: TECHNICS
- 2015-06-15: **New protocol**: ACP24
- 2015-05-29: **New protocol**: S100
- 2015-05-29: Some smaller corrections
- 2015-05-28: Added Logging for XMega
- 2015-05-28: Timing corrections for FAN protocol
- 2015-05-27: **New protocol**: MERLIN
- 2015-05-27: **New protocol**: FAN
- 2015-05-18: Added F_CPU macro for STM32L1XX
- 2015-05-18: Some corrections for XMega port
- 2015-04-23: **New protocol**: PENTAX
- 2015-04-23: Port to AVR XMega
- 2014-09-19: Bugfix: added missing newline before #else
- 2014-09-18: Added logging for ARM STM32F10X
- 2014-09-17: Corrected PROGMEM access to array irmp_protocol_names[].
- 2014-09-15: Changed timing tolerances for KASEIKYO protocol

- 2014-09-15: Moved irmp_protocol_names to flash, additional UART routines in irmp-main-avr-uart.c
- 2014-07-21: Port to PIC 12F1840
- 2014-07-09: **New protocol**: SAMSUNG48
- 2014-07-09: Some small corrections
- 2014-07-01: Added logging for ARM_STM32F4XX
- 2014-07-01: IRMP port for PIC XC8 compiler, removed variadic macros because of stupid XC8 compiler :-(
- 2014-06-05: **New protocol**: LGAIR
- 2014-05-30: **New protocol**: SPEAKER
- 2014-05-30: Optimized timings for SAMSUNG protocol
- 2014-02-20: Corrected decoding of SIEMENS protocol
- 2014-02-19: **New protocols**: RCMM32, RCMM24 and RCMM12
- 2014-09-17: Optimized timing for ROOMBA
- 2013-04-09: **New protocol**: ROOMBA
- 2013-04-09: Optimized detection of ORTEK (Hama) frames
- 2013-03-19: **New protocol**: ORTEK (Hama)
- 2013-03-19: **New protocol**: TELEFUNKEN
- 2013-03-12: Changed timing tolerancies for RECS80- and RECS80EXT protocol
- 2013-01-21: Corrected detection of repetition frame beim DENON protocol
- 2013-01-17: Corrected frame detection beim DENON protocol
- 2012-12-11: **New protocol**: A1TVBOX
- 2012-12-07: Improved detection von DENON repetition frame
- 2012-11-19: Port to Stellaris LM4F120 TI Launchpad (ARM Cortex M4)
- 2012-11-06: Corrected DENON frame detection
- 2012-10-26: Some timer corrections and adaptations for Arduino
- 2012-07-11: **New protocol**: BOSE
- 2012-06-18: Added support for ATtiny87/167
- 2012-06-05: Some smaller corrections of port to ARM STM32
- 2012-06-05: Correction of include in irmpextlog.c
- 2012-06-05: Bugfix, if only NEC and NEC42 activated
- 2012-05-23: Port to ARM STM32
- 2012-05-23: Bugfix frame detection for DENON protocol
- 2012-02-27: Bugfix in IR60-Decoder
- 2012-02-27: Bugfix in CRC calculation of KASEIKYO frames
- 2012-02-27: Port to C18 Compiler for PIC microcontrollers
- 2012-02-13: Bugfix: most significant bit in Address wrong in NEC protocol, if NEC42 protocol activated, too
- 2012-02-13: Corrected timing of SAMSUNG- and SAMSUNG32 protocol
- 2012-02-13: KASEIKYO: Genre2 bits will be now stored in upper nibble of flags
- 2011-09-20: **New protocol**: KATHREIN
- 2011-09-20: **New protocol**: RUWIDO
- 2011-09-20: **New protocol**: THOMSON
- 2011-09-20: **New protocol**: IR60 (SDA2008)
- 2011-09-20: **New protocol**: LEGO

- 2011-09-20: **New protocol**: NEC16
- 2011-09-20: **New protocol**: NEC42
- 2011-09-20: **New protocol**: NETBOX
- 2011-09-20: Port to ATtiny84 and ATtiny85
- 2011-09-20: Improved key repetition detection in RC5 protocol
- 2011-09-20: Improved decoding of Biphase protocols
- 2011-09-20: Fixed some smaller bugs in RECS80 decoder
- 2011-09-20: Corrected detection of additional bits in SIRCS protocol
- 2011-01-18: Some corrections for SIEMENS protocol
- 2011-01-18: **New protocol**: Nikon
- 2011-01-18: SIRCS: additional bits (>12) will be stored in address
- 2011-01-18: Some timing corrections for DENON protocol
- 2010-09-04: Bugfix for F_INTERRUPTS >= 16000
- 2010-09-02: **New protocol**: RC6A
- 2010-08-29: **New protocol**: JVC
- 2010-08-29: KASEIKYO protocol: genre bits will be now stored
- 2010-08-29: KASEIKYO protocol: Improved handling of repetition frames
- 2010-08-29: Improved support of APPLE protocols.
- 2010-07-01: Bugfix: added a timeout for NEC repetition frames. This avoids 'ghost commands'.
- 2010-06-26: Bugfix: deactivated RECS80, RECS80EXT & SIEMENS if interrupts frequency is low
- 2010-06-25: **New protocol**: RCCAR
- 2010-06-25: Extended keyboard detection for FDC protocol (IR keyboard)
- 2010-06-25: Interrupt frequency now up to 20kHz possible
- 2010-06-09: **New protocol**: FDC (IR-keyboard)
- 2010-06-09: Corrected timing for DENON protocol
- 2010-06-02: **New protocol**: SIEMENS (Gigaset)
- 2010-05-26: **New protocol**: NOKIA
- 2010-05-26: Bugfix: detection of long keyboard press for GRUNDIG protocol
- 2010-05-17: Bugfix SAMSUNG32 protocol: corrected command bit mask
- 2010-05-16: **New protocol**: GRUNDIG
- 2010-05-16: Improved handling of automatic frame repetitions for SIRCS-, SAMSUNG32-, and NUBERT protocol
- 2010-04-28: Only some cosmetic code optimizations
- 2010-04-16: Improved all timing tolerancies
- 2010-04-12: **New protocol**: Bang & Olufsen
- 2010-03-29: Bugfix: detection of multiple NEC repetition frames
- 2010-03-29: Moved configuration data to irmpconfig.h
- 2010-03-29: Introduced a program version in README.txt: Version 1.0
- 2010-03-17: **New protocol**: NUBERT
- 2010-03-16: Correction of RECS80 start bit timings
- 2010-03-16: **New protocol**: RECS80 Extended
- 2010-03-15: Some optimizations
- 2010-03-14: Port to PIC

- 2010-03-11: Some adjustements for some ATMegas
- 2010-03-07: Bugfix: Reset of state machine after a incomplete RC5 frame
- 2010-03-05: **New protocol**: APPLE
- 2010-03-05: Data irmp_data.addr + irmp_data.command will be now stored in the bit order of the appropriate protocol
- 2010-03-04: **New protocol**: SAMSUNG32 (Mix aus SAMSUNG & NEC protocol)
- 2010-03-04: Changed some timer tolerances changes of SIRCS- and KASEIKYO
- 2010-03-02: SIRCS: corrected detection and suppression of automatic frame repetitions
- 2010-03-02: SIRCS: device ID bits will be now stored in irmp_data.command (not irmp_data.address anymore)
- 2010-03-02: Enlargement of scan buffers (for logging)
- 2010-02-24: New variable flags in IRMP_DATA for detection of long key press
- 2010-02-20: Bugfix DENON protocol: repetition frame is now basically inverted
- 2010-02-19: Detection of NEC protocol-Varianten, z. B. APPLE-Fernbedienung
- 2010-02-19: Detection of RC6- and DENON protocol
- 2010-02-19: Some improvements for RC5 decoders (Bugfixes)
- 2010-02-13: Bugfix: Puls/Pause counters were 1 too low, now better detection of protokols with very short pulses
- 2010-02-13: Improved detection of NEC repetition frames
- 2010-02-12: New: RC5 protocol
- 2010-02-05: Eliminated a conflict between SAMSUNG- and MATSUSHITA protocol
- 2010-01-07: First version

Literature

IR Abstract

- http://www.sbprojects.net/knowledge/ir/index.php
- http://www.epanorama.net/links/irremote.html
- http://www.elektor.de/jahrgang/2008/juni/cc2-avr-projekt-%283%29-unsichtbare-kommandos.497184.lynkx?tab=4
- http://mc.mikrocontroller.com/de/IR-Protokolle.php

SIRCS Protocol

- http://www.sbprojects.net/knowledge/ir/sirc.php
- http://mc.mikrocontroller.com/de/IR-Protokolle.php#SIRCS
- http://www.ustr.net/infrared/sony.shtml
- http://users.telenet.be/davshomepage/sony.htm
- http://picprojects.org.uk/projects/sirc/
- http://www.celadon.com/infrared_protocol/infrared_protocols_samples.pdf

NEC Protocol

- http://www.sbprojects.net/knowledge/ir/nec.php

- http://www.ustr.net/infrared/nec.shtml
- http://www.celadon.com/infrared_protocol/infrared_protocols_samples.pdf

ACP24 Protocol

The ACP24-Protocol is used by Stiebel-Eltron-Aircons

The structure of the 70 databits is :

```
             1         2         3         4         5         6

01234567890123456789012345678901234567890123456789012345678901234567890123456789
   N VVMMM    ? ???    t vmA x                 y
TTTT

00110010000001110000100010100000000000000000100000000000000000000000001111on
, temp=30
```

These are converted into the following 16 bits from irmp_data.command:

```
       5432109876543210
       NAVVvMMMmtxyTTTT
```

Meaning of the symbols:

```
   TTTT = Temperature + 15 degree
          TTTT
          ----------
          0000        ???
          0001        ???
          0010        ???
          0011        18 degree
          0100        19 degree
          0101        20 degree
          0110        21 degree
          ...
          1111        30 degree

   N    = Nightmode
          N
          ----------
          0           off
          1           on

   VV   = fan, v must be 1!
          VV   v
          ----------
          00   1      level 1
          01   1      level 2
          10   1      level 3
          11   1      Automatic

   MMM  = Mode
          MMM  m
```

```
          ----------
          000  0      turn off
          001  0      turn on
          001  1      cooling
          010  1      fan
          011  1      demist
          100  1      ???
          101  1      ---
          110  1      ---
          111  1      ---

   A    = Automatic-Programm
          A
          ----------
          0           off
          1           on

   t    = Timer
          t   x y
          ----------
          1   1 0     Timer 1
          1   0 1     Timer 2
```

To control the air con via IRSND, the following functions can be used:

```
#include "irmp.h"
#include "irsnd.h"

#define IRMP_ACP24_TEMPERATURE_MASK         0x000F
// TTTT

#define IRMP_ACP24_SET_TIMER_MASK           (1<<6)
// t
#define IRMP_ACP24_TIMER1_MASK              (1<<5)
// x
#define IRMP_ACP24_TIMER2_MASK              (1<<4)
// y

#define IRMP_ACP24_SET_MODE_MASK            (1<<7)
// m
#define IRMP_ACP24_MODE_POWER_ON_MASK       (1<<8)
// MMMm = 0010 Einschalten
#define IRMP_ACP24_MODE_COOLING_MASK        (IRMP_ACP24_SET_MODE_MASK | (1<<8))
// MMMm = 0011 Kuehlen
#define IRMP_ACP24_MODE_VENTING_MASK        (IRMP_ACP24_SET_MODE_MASK | (1<<9))
// MMMm = 0101 Lueften
#define IRMP_ACP24_MODE_DEMISTING_MASK      (IRMP_ACP24_SET_MODE_MASK | (1<<10) |
(1<<8))   // MMMm = 1001 Entfeuchten

#define IRMP_ACP24_SET_FAN_STEP_MASK        (1<<11)
// v
#define IRMP_ACP24_FAN_STEP_MASK            0x3000
// VV
#define IRMP24_ACP_FAN_STEP_BIT             12
// VV
#define IRMP_ACP24_AUTOMATIC_MASK           (1<<14)
// A
#define IRMP_ACP24_NIGHT_MASK               (1<<15)
// N
```

```
// possible values for acp24_set_mode();
#define ACP24_MODE_COOLING                  1
#define ACP24_MODE_VENTING                  2
#define ACP24_MODE_DEMISTING                3

static uint8_t temperature = 18;
// 18 degrees

static void
acp24_send (uint16_t cmd)
{
    IRMP_DATA irmp_data;

    cmd |=  (temperature - 15) & IRMP_ACP24_TEMPERATURE_MASK;

    irmp_data.protocol = IRMP_ACP24_PROTOCOL;
    irmp_data.address  = 0x0000;
    irmp_data.command  = cmd;
    irmp_data.flags    = 0;

    irsnd_send_data (&irmp_data, 1);
}

void
acp24_set_temperature (uint8_t temp)
{
    uint16_t    cmd = IRMP_ACP24_MODE_POWER_ON_MASK;

    temperature = temp;
    acp24_send (cmd);
}

void
acp24_off (void)
{
    uint16_t    cmd = 0;
    acp24_send (cmd);
}

#define ACP_FAN_STEP1       0
#define ACP_FAN_STEP2       1
#define ACP_FAN_STEP3       2
#define ACP_FAN_AUTOMATIC   3

void
acp24_fan (uint8_t fan_step)
{
    uint16_t    cmd = IRMP_ACP24_MODE_POWER_ON_MASK;

    cmd |= IRMP_ACP24_SET_FAN_STEP_MASK | ((fan_step << IRMP24_ACP_FAN_STEP_BIT) &
IRMP_ACP24_FAN_STEP_MASK);

    acp24_send (cmd);
}

void
acp24_set_mode (uint8_t mode)
{
    uint16_t    cmd = 0;

    switch (mode)
```

```
        {
            case ACP24_MODE_COOLING:    cmd = IRMP_ACP24_MODE_COOLING_MASK;     break;
            case ACP24_MODE_VENTING:    cmd = IRMP_ACP24_MODE_VENTING_MASK;     break;
            case ACP24_MODE_DEMISTING:  cmd = IRMP_ACP24_MODE_DEMISTING_MASK;   break;
            default: return;
        }
        acp24_send (cmd);
    }

    void
    acp24_program_automatic (void)
    {
        uint16_t    cmd = IRMP_ACP24_MODE_POWER_ON_MASK | IRMP_ACP24_AUTOMATIC_MASK;
        acp24_send (cmd);
    }

    void
    acp24_program_night (void)
    {
        uint16_t    cmd = IRMP_ACP24_MODE_POWER_ON_MASK | IRMP_ACP24_NIGHT_MASK;
        acp24_send (cmd);
    }
```

LGAIR Protocol

The LG Air Con is controlled by an 'intelligent' remote. These are the encoded data:

```
   Command                 AAAAAAAA  PW  Z  S  T  mmm  tttt  vvvv  PPPP
   ------------------------------------------------------------------------
   ON 23C                  10001000  00  0  0  0  000  1000  0100  1100
   ON 26C                  10001000  00  0  0  0  000  1011  0100  1111

   OFF                     10001000  11  0  0  0  000  0000  0101  0001
   TURN OFF                10001000  11  0  0  0  000  0000  0101  0001
(18C currently, identical to off)

   TEMP DOWN 23C           10001000  00  0  0  1  000  1000  0100  0100
   MODE (to mode0, 23C)    10001000  00  0  0  1  000  1000  0100  0100

   TEMP UP (24C)           10001000  00  0  0  1  000  1001  0100  0101
   TEMP DOWN 24C           10001000  00  0  0  1  000  1001  0100  0101

   TEMP UP (25C)           10001000  00  0  0  1  000  1010  0100  0110
   TEMP DOWN 25C           10001000  00  0  0  1  000  1010  0100  0110

   TEMP UP (26C)           10001000  00  0  0  1  000  1011  0100  0111

   MODE                    10001000  00  0  0  1  011  0111  0100  0110
(to mode1, 22C - when switching to mode1 temp automaticall sets to 22C)
   ON (mode1, 22C)         10001000  00  0  0  0  011  0111  0100  1110

   MODE                    10001000  00  0  0  1  001  1000  0100  0101
(to mode2, no temperature displayed)
   ON (mode2)              10001000  00  0  0  0  001  1000  0100  1101
   MODE (to mode3, 23C)    10001000  00  0  0  1  100  1000  0100  1000
   ON (mode3, 23C)         10001000  00  0  0  0  100  1000  0100  0000
```

VENTILATION SLOW	10001000	00	0	0	1	000	0011	0000	1011
VENTILATION MEDIUM	10001000	00	0	0	1	000	0011	0010	1101
VENTILATION HIGH	10001000	00	0	0	1	000	0011	0100	1111
VENTILATION LIGHT	10001000	00	0	0	1	000	0011	0101	0000
SWING ON/OFF	10001000	00	0	1	0	000	0000	0000	0001

Format: 1 start bit + 8 address bits + 16 data bits + 4 checksum bits + 1 stop bit

Address: AAAAAAAA = 0x88 (8 bits)

Data: PW Z S T MMM tttt vvvv PPPP (16 bits)

PW: Power: 00 = On, 11 = Off

Z: N/A: Always 0

S: Swing: 1 = Toggle swing, all other data bits are zeros.

T: Temp/Vent: 1 = Set temperature and ventilation

MMM: Mode, can be combined with temperature
000=Mode 0
001=Mode 2
010=????
011=Mode 1
100=Mode 3
101=???
111=???

tttt: Temperature:
0000=used by OFF command
0001=????
0010=????
0011=18°C
0100=19°C
0101=20°C
0110=21°C
0111=22°C
1000=23°C
1001=24°C
1010=25°C
1011=26°C
1011=27°C
1100=28°C
1101=29°C
1111=30°C

vvvv: Ventilation:
0000=slow
0010=medium

```
0011=????
0100=high
0101=light
0110=????
0111=????
...
1111=????
```

```
   Checksum:   PPPP = (DataNibble1 + DataNibble2 + DataNibble3 +
DataNibble4) & 0x0F
```

NEC16 Protocol (JVC)

- http://www.sbprojects.net/knowledge/ir/jvc.php
- http://www.ustr.net/infrared/jvc.shtml

Samsung Protocol

(was reverse engineered by several protocols (Daewoo or similar), so no direct link to Samsung documents is available)

Here is a link to the Daewoo-protocol, which uses the same principle of the sync-bits in the center of a frame, but words with different timings:

- http://users.telenet.be/davshomepage/daewoo.htm

MATSUHITA Protocol

- http://www.celadon.com/infrared_protocol/infrared_protocols_samples.pdf

KASEIKYO Protocol ("Japan Protocol")

- http://www.mikrocontroller.net/attachment/4246/IR-Protokolle_Diplomarbeit.pdf
- http://www.roboternetz.de/phpBB2/files/entwicklung_und_realisierung_einer_universalinfrarotfernbedienung_mit_timerfunktionen.pdf

RECS80 and RECS80 Extended Protocol

- http://www.sbprojects.net/knowledge/ir/recs80.php

RC5 and RC5x Protocol

- http://www.sbprojects.net/knowledge/ir/rc5.php
- http://mc.mikrocontroller.com/de/IR-Protokolle.php#RC5
- http://users.telenet.be/davshomepage/rc5.htm
- http://www.celadon.com/infrared_protocol/infrared_protocols_samples.pdf
- http://www.opendcc.de/info/rc5/rc5.html

Denon Protocol

- http://www.mikrocontroller.com/de/IR-Protokolle.php#DENON
- http://www.manualowl.com/m/Denon/AVR-3803/Manual/170243

RC6 and RC6A Protocol

- http://www.sbprojects.net/knowledge/ir/rc6.php
- http://www.picbasic.nl/info_rc6_uk.htm

Bang & Olufsen

- http://www.mikrocontroller.net/attachment/33137/datalink.pdf

Grundig Protocol

- http://www.see-solutions.de/sonstiges/Grundig_10bit.pdf

Nokia Protocol

- http://www.sbprojects.net/knowledge/ir/nrc17.php

IR60 (SDA2008 and MC14497P)

- http://www.datasheetcatalog.org/datasheet/motorola/MC14497P.pdf

LEGO Power Functions RC

- http://www.philohome.com/pf/LEGO_Power_Functions_RC_v110.pdf

RCMM Protocol

- http://www.sbprojects.net/knowledge/ir/rcmm.php

Other Protocols

- http://www.mikrocontroller.net/attachment/4246/IR-Protokolle_Diplomarbeit.pdf
- http://www.celadon.com/infrared_protocol/infrared_protocols_samples.pdf
- http://www.roboternetz.de/phpBB2/files/entwicklung_und_realisierung_einer_universalinfrarotfernbedienung_mit_timerfunktionen.pdf

IRMP on Youtube

- http://www.youtube.com/watch?v=Q7DJvLIyTEI
- http://www.youtube.com/watch?v=1tQ_aqayWZk
- http://www.youtube.com/watch?v=W4tI2axR3-w
- http://www.youtube.com/watch?v=SRs98dIe2WE

Other Artikels

Whitepaper von Martin Gotschlich, Infineon Technologies AG

Hardware / IRMP Projects

Remote IRMP

Infrared sender und receiver controlled via ip network with Android smartphone as remote control:

* http://www.mikrocontroller.net/articles/Remote_IRMP

IR Tester

IR tester with LCD by Klaus Leidinger:

- http://www.mikrocontroller-projekte.de/Mikrocontroller/index.html

IR Tester with AVR-NET-IO

IR tester for Pollin AVR-NET-IO with Pollin ADD-ON Board:

- http://son.ffdf-clan.de/include.php?path=forumsthread&threadid=703

USB IR Remote Receiver

USB IR remote receiver by Hugo Portisch:

- http://www.mikrocontroller.net/articles/USB_IR_Remote_Receiver

USB IR Receiver/Sender/Switch with Wakeup-Timer

- http://www.vdr-portal.de/board18-vdr-hardware/board13-fernbedienungen/123572-fertig-irmp-auf-stm32-ein-usb-ir-empf%C3%A4nger-sender-einschalter-mit-wakeup-timer/
- http://www.mikrocontroller.net/articles/IRMP_auf_STM32_-_ein_USB_IR_Empf%C3%A4nger/Sender/Einschalter_mit_Wakeup-Timer

USBASP

IR switch based on USBasp

- http://wiki.easy-vdr.de/index.php?title=USBASP_Einschalter

Servo controlled IR Sender

Servo controlled IR Sender (adaptive) by Stefan Pendsa:

- http://forum.mikrokopter.de/topic-21060.html
- SVN

Adaptive IR Remote Control

Adaptive IR remote control by Robert and Frank M.

- http://www.mikrocontroller.net/articles/DIY_Lernfähige_Fernbedienung_mit_IRMP

AVR Moodlight

AVR Moodlight by Axel Schwenke

- http://www.mikrocontroller.net/topic/244768

STM8 Moodlight by Axel Schwenke

- https://www.mikrocontroller.net/topic/380098

Infinity Mirror LED Ceiling Lamp

Infinity Mirror LED ceiling lamp with remote control by Philipp Meißner

- http://digital-nw.de/Infinity-Mirror.htm

Cinema Control

Cinema control by Owagner

- http://ccc.zerties.org/index.php/Benutzer:Owagner

Leading-Edge Control

leading-edge control:

- http://flosserver.dyndns.org/phasenanschnittsdimmer.php

IRDioder – Ikea Dioder Hack

Ikea Dioder Hack:

- http://marco-difeo.de/tag/infrared/

Expedit Coffee Bar

Ikea Expedit as coffee bar:

- http://chaozlabs.blogspot.de/2013/09/expedit-coffee-bar.html

Arduino as IR Receiver

Arduino as IR Receiver:

- http://www.vdr-portal.de/board18-vdr-hardware/board13-fernbedienungen/110918-arduino-als-ir-empf%C3%A4nger-einsetzen/

More example from the Arduino library:

- https://github.com/ukw100/IRMP/tree/master/examples

IR Volume Control with Stellaris Launchpad

volume control with Stellaris Launchpad (ARM Cortex-M4F):

- http://www.anthonyvh.com/2013/03/31/ir-volume-control/

RemotePi Board

Shutdown RaspPI with IR remote control:

- http://www.msldigital.com/pages/more-information

Ethernut & IRMP

IRMP under RTOS Ethernut:

- http://www.klkl.de/ethernut.html

LED strip Remote Control

LED strip remote control:

- http://www.solderlab.de/index.php/misc/led-strip-remote-control

ADAT Audio Mixer

Audio Mixer:

- http://mailtonne.de/adat-audio-mixer/

Ethersex & IRMP

IRMP + IRSND Modul in Ethersex, a modular Firmware for AVR MCUs

- http://ethersex.de/index.php/IRMP

Mastermind Solver

Mastermind solver with LED stripes and IR remote control:

- http://www.mystrobl.de/Plone/basteleien/weitere-bulls-and-cows-mastermind-implementationen/mm-v1821/mastermind-solver-mit-led-streifen-und-ir-fernbedienung

A MythTV Remote Control without LIRC

PC Remote Control with ATtiny85

- http://tomscircuits.blogspot.de/2014/12/a-mythtv-remote-control-without-lirc.html

IRMP + IRSND Library for STM32F4

IRMP for STM32F4

- http://mikrocontroller.bplaced.net/wordpress/?page_id=1516

IRSND for STM32F4

- http://mikrocontroller.bplaced.net/wordpress/?page_id=1940

IRMP on STM32 - Construction Guidance

- http://www.mikrocontroller.net/articles/IRMP_auf_STM32_-_Bauanleitung

Seminar Paper - Extension of Arduino Platform

- www.eislab.fim.uni-passau.de/files/publications/2010/StudentDiener_ErweiterungDerArduinoPlattform.pdf

Acknowledgment

I thank Vlad Tepesch, Klaus Leidinger, Peter K., and Dániel Körmendi, who sent me many scan files of their IR remote controls.

I thank Christian F. for his tipps relating to the port to PIC MCUs, gera for the port to PIC-C18 compiler, kichi (Michael K.) for the port to ARM STM32, Markus Schuster for the port to TI's Stellaris LM4F120 Launchpad (ARM Cortex M4), Matthias Frank for the port to XMega, Wolfgang S. for the port to ESP8266, Achill Hasler for the port to Teensy, Axel Schwenke for the port to STM8.

Thanks to Dániel Körmendi, who added the LG-AIR protocol to IRSND. Thanks to Ulrich v.d. Kammer for the Pentax protocol extension in IRSND.

At least I will thank Jojo S. and Antonio T. for their great job translating and revisioning this documentation!

Discussion

You can discuss IRMP & IRSND in the German thread Infrared Multi Protocol Decoder.

Have fun with IRMP!

Kategorien:

- Infrarot
- AVR-Projekte