IRMP

Von **Frank M. (ukw (http://www.mikrocontroller.net/user/show/ukw))**

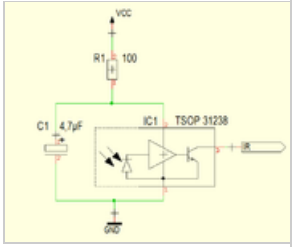
Da RC5 nicht nur veraltet, sondern mittlerweile obsolet ist und immer mehr die elektronischen Geräte der fernöstlichen Unterhaltungsindustrie in unseren Haushalten Einzug finden, ist es an der Zeit, einen IR-Decoder zu entwickeln, der ca. 90% aller bei uns im täglichen Leben zu findenden IR-Fernbedienungen "versteht".

Im folgenden wird IRMP als "Infrarot-Multiprotokoll-Decoder" in allen Einzelheiten vorgestellt. Auch das Gegenstück, nämlich IRSND als IR-Encoder, wird in diesem Artikel behandelt.

Inhaltsverzeichnis

IRMP - Infrarot-Multiprotokoll-Decoder

Unterstützte µCs



Anschluß eines IR-Empfängers an µC

IRMP ist u.a. lauffähig auf folgenden AVR µCs:

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

Es gibt aber auch Portierungen auf diverse PIC µCs - für den CCS- und C18-Compiler. Auch ist IRMP mittlerweile auf ARM STM32 und Stellaris LM4F120 Launchpad von TI (ARM Cortex M4) lauffähig.

Unterstützte IR-Protokolle

IRMP - der Infrarot-Fernbedienungsdecoder, der mehrere Protokolle auf einmal decodieren kann, beherrscht folgende Protokolle (in alphabetischer Reihenfolge):

Unterstützte Protokolle

Protokoll	Hersteller
A1TVBOX	ADB (Advanced Digital Broadcast), z.B. A1 TV Box
APPLE	Apple
B&O	Bang & Olufsen
BOSE	Bose
DENON	Denon, Sharp
FDC	FDC Keyboard
GRUNDIG	Grundig
NOKIA	Nokia, z.B. D-Box
IR60 (SDA2008)	Diverse europäische Hersteller
JVC	JVC
KASEIKYO	Panasonic, Technics, Denon und andere japanische Hersteller, welche Mitglied der "Japan's Association for Electric Home Application" sind.
KATHREIN	KATHREIN
LEGO	Lego
LGAIR	LG Air Conditioner
MATSUSHITA	Matsushita
NEC16	JVC, Daewoo
NEC42	JVC
NEC	NEC, Yamaha, Canon, Tevion, Harman/Kardon, Hitachi, JVC, Pioneer, Toshiba, Xoro, Orion, NoName und viele weitere japanische Hersteller.

NETBOX	Netbox
NIKON	NIKON
NUBERT	Nubert, z.B. Subwoofer System
ORTEK	Ortek, Hama
RC5	Philips und andere europäische Hersteller
RC6A	Philips, Kathrein und andere Hersteller, z.B. XBOX
RC6	Philips und andere europäische Hersteller
RCCAR	RC Car: IR Fernbedienung für Modellfahrzeuge
RECS80	Philips, Nokia, Thomson, Nordmende, Telefunken, Saba
RECS80EXT	Philips, Technisat, Thomson, Nordmende, Telefunken, Saba
RCMM	Fujitsu-Siemens z.B. Activy keyboard **(NEU!)**
ROOMBA	iRobot Roomba Staubsauger
SAMSUNG32	Samsung
SAMSUNG48	Div. Klimaanlagen Hersteller (**(NEU!)**)
SAMSUNG	Samsung
RUWIDO	RUWIDO (z.B. T-Home-Mediareceiver, MERLIN-Tastatur (Pollin))
SIEMENS	Siemens, z.B. Gigaset M740AV
SIRCS	Sony
SPEAKER	Lautsprecher Systeme wie z.B. X-Tensions **(NEU!)**
TELEFUNKEN	Telefunken
THOMSON	Thomson

Jedes dieser Protokolle ist einzeln aktivierbar. Wer möchte, kann alle Protokolle aktivieren. Wer nur ein Protokoll braucht, kann alle anderen deaktivieren. Es wird nur das vom Compiler übersetzt, was auch benötigt wird.

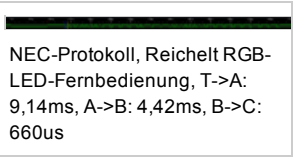
Entstehung

Der auf AVR- und PIC-µCs einsetzbare Source zu IRMP entstand im Rahmen des Word Clock Projektes.

Thread im Forum

Anlass für einen eigenen IRMP-Artikel ist folgender Thread in der Codesammlung: Beitrag: IRMP - Infrared Multi Protocol Decoder (http://www.mikrocontroller.net/topic/162119)

IR-Protokolle



NEC-Protokoll, Reichelt RGB-LED-Fernbedienung, T->A: 9,14ms, A->B: 4,42ms, B->C: 660us

Einige Hersteller verwenden ihr eigenes hausinterne Protokoll, dazu gehören u.a. Sony, Samsung und Matsushita. Philips hat RC5 entwickelt und natürlich auch selbst benutzt. RC5 galt damals in Europa als *das* Standard-IR-Protokoll, welches von vielen europäischen Herstellern übernommen wurde. Mittlerweile ist RC5 fast gar nicht mehr anzutreffen - man kann es eigentlich als "ausgestorben" abhaken. Der Nachfolger RC6 wird zwar noch in einigen aktuellen europäischen Geräten eingesetzt, ist aber auch nur vereinzelt vorzufinden.

Auch die japanischen Hersteller haben versucht, einen eigenen Standard zu etablieren, nämlich das sog. Kaseikyo- (oder auch "Japan-") Protokoll. Dieses ist mit einer Bitlänge von 48 sehr universell und allgemein verwendbar. Richtig durchgesetzt hat es sich aber bis heute nicht - auch wenn man es hier und da im heimischen Haushalt vorfindet.

Heutzutage wird (auch vornehmlich bei japanischen Geräten) das NEC-Protokoll verwendet - und zwar von den unterschiedlichsten (Marken- und auch Noname-)Herstellern. Ich schätze den "Marktanteil" auf ca. 80% beim NEC-Protokoll. Fast alle Fernbedienungen im alltäglichen Einsatz verwenden bei mir den NEC-IR-Code. Das fängt beim Fernseher an, geht über vom DVD-Player zur Notebook-Fernbedienung und reicht bis zur Noname-MultiMedia-Festplatte - um nur einige Beispiele zu nennen.

Kodierungen

IRMP unterstützt folgende IR-Codings:

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

Die Pulse werden dabei moduliert - üblicherweise mit 36kHz oder 38kHz - um Umwelteinflüsse wie Raum- oder Sonnenlicht ausfiltern zu können.

Pulse Distance

Eine Pulse Distance Kodierung erkennt man an der folgenden Regel:

- es gibt nur **eine Pulslänge** und **zwei verschiedene Pausenlängen.**

Pulse Width


Pulse Distance Coding

Bei der Pulse Width Kodierung gilt die Regel:

- es gibt **zwei verschiedene Pulslängen** und nur **eine Pausenlänge**

Pulse Distance Width


Pulse Width Coding

Dies ist ein Mischmasch aus Pulse Distance und Pulse Width Coding.

Also:


Pulse Distance Width Coding

- es gibt **zwei verschiedene Pulslängen** und **zwei verschiedene Pausenlängen.**

Biphase

Bei der Biphase Kodierung entscheidet die Reihenfolge von Puls und Pause über den Wert des Bits.


Biphase Coding

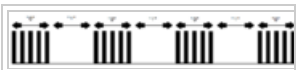
Damit erkennt man ein Biphase-Coding an folgendem Kriterium:

- es kommen genau **eine** Pausen- und eine Pulslänge, sowie jeweils die **doppelten** Puls-/Pausenlängen vor

Normalerweise sind die Längen für die Pulse und Pausen gleich, d.h. die Signalform ist symmetrisch. IRMP erkennt aber auch Protokolle, die mit unterschiedlichen Puls-/Pause-Längen arbeiten. Dies ist zum Beispiel bei dem A1TVBOX-Protokoll der Fall.

Pulse Position

Die Pulse Position Kodierung kennt man von den üblichen UARTs. Hier hat jedes Bit eine feste Länge. Je nach Wert (0 oder 1) ist es ein Puls oder eine Pause.


Pulse Position Coding

Typisches Kriterium für ein **Pulse Position Protokoll** ist:

- es kommen **Vielfache** einer Grund-Puls-/Pausenlänge vor

Eine tabellarische Aufstellung der verschiedenen IR-Protokolle findet man hier: Die IR-Protokolle im Detail.

Die dort angegebenen Timingwerte sind Idealwerte. Bei einigen Fernbedienungen in der Praxis weichen sie um bis zu 40% voneinander ab. Deshalb arbeitet IRMP mit Minimum-/Maximumsgrenzen, um bzgl. des Zeitverhaltens tolerabel zu sein.

Protokoll-Erkennung

Die von IRMP decodierten Protokolle haben etwas gemeinsames: Sie weisen alle ein Start-Bit auf, welches vom Timing her ausgezeichnet, d.h. einmalig ist.

Anhand dieses Start-Bit-Timings werden die verschiedenen Protokolle unterschieden. IRMP misst also das Timing des Start-Bits und stellt dann "on-thy-fly" seine Timingtabellen auf das erkannte Protokoll um, damit die nach dem Start-Bit gesandten Daten in einem Rutsch eingelesen werden können, ohne das komplette Telegramm (Frame) erst speichern zu müssen. IRMP wartet also nicht darauf, dass ein kompletter Frame eingelesen wurde, sondern legt direkt nach der ersten Pulserkennung los.

Ist das gelesene Start-Bit nicht eindeutig, fährt IRMP "mehrspurig", d.h. es werden zum Beispiel zwei mögliche Protokolle gleichzeitig verfolgt. Sobald aus Plausibilitätsgründen eines der beiden Protokolle nicht mehr möglich sein kann, wird komplett auf das andere Protokoll gewechselt.

Realisiert wird die Erkennung über eine Statemachine, die timergesteuert über eine Interruptroutine in regelmäßigen Abständen (üblicherweise 15.000 mal in der Sekunde) aufgerufen wird. Die Statemachine kennt (unter anderem) folgende Zustände:

- Erkenne den ersten Puls des Start-Bits
- Erkenne die Pause des Start-Bits
- Erkenne den Puls des ersten Datenbits

Danach sind die Puls/Pause-Längen des Startbits bekannt. Nun werden alle vom Anwender aktivierten Protokolle nach diesen Längen durchsucht. Wurde ein Protokoll gefunden, werden die Timing-Tabellen dieses Protokolls geladen und im weiteren geprüft, ob die nachfolgenden Puls-/Pause-Zeiten innerhalb der geladenen Werte übereinstimmen.

Es geht also weiter in der Statemachine mit folgenden Zuständen

- Erkenne die Pausen der Datenbits
- Erkenne die Pulse der Datenbits
- Prüfe Timing. Wenn abweichend, schalte um auf ein anderes noch in Frage kommendes IR-Protokoll, ansonsten schalte Statemachine komplett zurück
- Erkenne das Stop-Bit, falls das Protokoll eines vorsieht
- Prüfe Daten auf Plausibilität, wie CRC oder andere redundante Datenbits
- Wandle die Daten in Geräte-Adresse und Kommando
- Erkenne Wiederholungen durch längere Tastendrücke, setze entsprechendes Flag

Tatsächlich ist die Statemachine noch etwas komplizierter, da manche Protokolle gar kein Start-Bit (z.B. Denon) bzw. mehrere Start-Bits (z.B. 4 bei B&O) haben bzw. mitten im Frame ein weiteres Synchronisierungs-Bit (z.B. Samsung) vorsehen. Diese besonderen Bedingungen werden durch protokollspezifische "Spezialbehandlungen" im Code abgefangen.

Das Umschalten auf ein anderes Protokoll kann mehrfach während des Empfangs des Frames geschehen, z.B. von NEC42 (42 Bit) auf NEC16 (8 Bit + Sync-Bit + 8 Bit), wenn vorzeitig ein zusätzliches Synchronisierungsbit erkannt wurde, oder von NEC/NEC42 (32/42 Bit) auf JVC (16 Bit), wenn das Stop-Bit vorzeitig auftrat. Schwierig wird es dann, wenn zwei mögliche Protokolle nach Erkennung des Start-Bits unterschiedliche Kodierungen verwenden, z.B. wenn das eine Protokoll ein Pulse Distance Coding und das andere ein Biphase Coding (Manchester) benutzt. Hier speichert IRMP die jeweils völlig verschieden ermittelten Bits für beide Codierungen, um dann später die einen oder anderen Werte wieder zu verwerfen.

Desweiteren senden einige Fernbedienungen bei bestimmten Protokollen aus Gründen der Redundanz (Fehlererkennung) oder wegen längeren Tastendrucks Wiederholungsframes. Diese werden von IRMP unterschieden: Die für die Fehlererkennung zuständigen Frames werden von IRMP geprüft, aber nicht an die Anwendung zurückgegeben, die anderen werden als langer Tastendruck erkannt und entsprechend von IRMP gekennzeichnet.

Download

Version 2.6.7, Stand vom 19.09.2014

Download Release-Version: Irmp.zip (http://www.mikrocontroller.net/wikifiles/7/79/Irmp.zip)

IRMP & IRSND sind nun auch über SVN abrufbar: IRMP im SVN (http://www.mikrocontroller.net/svnbrowser/irmp/), Download Tarball (http://www.mikrocontroller.net/svnbrowser/irmp/?view=tar)

Achtung:

Die Version im SVN kann eine Zwischen- oder Test-Version sein, die nicht den hier dokumentierten Stand widerspiegelt! Im Zweifel verwendet man besser den obigen Download-Link auf Irmp.zip.

Die Software-Änderungen kann man sich hier anschauen: Software-Historie IRMP (http://www.mikrocontroller.net/articles/IRMP#Software-Historie_IRMP)

Source-Code

Der Source-Code lässt sich einfach für AVR-µCs übersetzen, indem man unter Windows die Projekt-Datei irmp.aps in das AVR Studio 4 lädt.

Für andere Entwicklungsumgebungen ist leicht ein Projekt bzw. Makefile angelegt. Zum Source gehören:

- irmp.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmp.c?view=markup) - Der eigentliche IR-Decoder
- irmpprotocols.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpprotocols.h?view=markup) - Sämtliche Definitionen zu den IR-Protokollen
- irmpsystem.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpsystem.h?view=markup) - Vom Zielsystem abhängige Definitionen für AVR/PIC/STM32
- irmp.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmp.h?view=markup) - Include-Datei für die Applikation
- irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) - Anzupassende Konfigurationsdatei

- main.c (http://www.mikrocontroller.net/svnbrowser/irmp/main.c?view=markup) - Beispiel Anwendung

WICHTIG

Im Applikations-Source sollte nur irmp.h per include eingefügt werden, also lediglich:

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

Alle anderen Include-Dateien werden automatisch über irmp.h "eingefügt". Siehe dazu auch die Beispieldatei main.c.

Desweiteren muss die Preprocessor-Konstante **F_CPU im Projekt bzw. Makefile** gesetzt werden. Diese sollte mindestens den Wert 8000000UL haben, der Prozessor sollte also zumindest mit 8 MHz laufen.

Auch auf PIC-Prozessoren ist IRMP lauffähig. Für den PIC-CCS-Compiler sind entsprechende Preprocessor-Konstanten bereits gesetzt, so dass man irmp.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmp.c?view=markup) direkt in der CCS-Entwicklungsumgebung verwenden kann. Lediglich eine kleine Interrupt-Routine wie

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

ist hinzuzufügen, wobei man den Interrupt auf 66µs (also 15kHz) stellt.

Für AVR-Prozessoren ist ein Beispiel für die Anwendung von IRMP in main.c (http://www.mikrocontroller.net/svnbrowser/irmp/main.c?view=markup) zu finden - im wesentlichen geht es da um die Timer-Initialisierung und den Abruf der empfangenen IR-Telegramme. Das empfangene Protokoll, die Geräte-Adresse und der Kommando-Code wird dann in der AVR-Version auf dem HW-UART ausgegeben.

Für das Stellaris LM4F120 Launchpad von TI (ARM Cortex M4) ist eine entsprechende Timer-Initialisierungsfunktion in main.c (http://www.mikrocontroller.net/svnbrowser/irmp/main.c?view=markup) bereits integriert.

Ebenso kann IRMP auf STM32-Mikroprozessoren eingesetzt werden.

avr-gcc-Optimierungen

Ab Version avr-gcc 4.7.x kann die LTO-Option (https://gcc.gnu.org/onlinedocs/gccint/LTO.html#LTO) genutzt werden, um den Aufruf der externen Funktion irmp_ISR() aus der eigentlichen ISR effizienter zu machen. Das verbessert das Zeitverhalten der ISR etwas.

Zu den sonst schon üblichen Compiler- und Linker-Optionen kommen noch folgende dazu:

- Zusätzliche Compiler-Option: -flto
- Zusätzliche Linker-Optionen: -flto -Os

Vergisst man (unter Windows?) die zusätzliche Linker-Option -Os, wird das Binary allerdings wesentlich größer, da dann nicht mehr optimiert wird. Auch muss -flto an den Linker übergeben werden, weil sonst die LTO-Optimierung nicht mehr greift.

Konfiguration

Die Konfiguration von IRMP wird über Parameter in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) vorgenommen, nämlich:

- Anzahl Interrupts pro Sekunde
- Unterstützte IR-Protokolle
- Hardware-Pin zum IR-Empfänger
- IR-Logging

Einstellungen in irmpconfig.h

IRMP decodiert sämtliche oben aufgelisteten Protokolle in einer ISR. Dafür sind einige Angaben nötig. Diese werden in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) eingestellt.

F_INTERRUPTS

Anzahl der Interrupts pro Sekunde. Der Wert kann zwischen 10000 und 20000 eingestellt werden. Je höher der Wert, desto besser die Auflösung und damit die Erkennung. Allerdings erkauft man sich diesen Vorteil mit erhöhter CPU-Last. Der Wert 15000 ist meist ein guter Kompromiss.

Standardwert:

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

IRMP_SUPPORT_xxx_PROTOCOL

Hier lässt sich einstellen, welche Protokolle von IRMP unterstützt werden sollen. Die Standardprotokolle sind bereits aktiv. Möchte man weitere Protokolle einschalten bzw. einige aus Speicherplatzgründen deaktivieren, sind die entsprechenden Werte in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) anzupassen.

```
// 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
```

Jedes von IRMP unterstützte IR-Protokoll "verbrät" ungefähr den oben angegebenen Speicher an Code. Hier kann man Optimierungen vornehmen: Zum Beispiel ist die Modulationsfrequenz von 455kHz beim B&O-Protokoll weitab von den Frequenzen, die von den anderen Protokollen verwendet werden. Hier braucht man evtl. andere IR-Empfänger, anderenfalls kann man diese Protokolle einfach deaktiveren. Zum Beispiel kann man mit einem TSOP1738 kein B&O-Protokoll (455kHz) mehr empfangen.

Ausserdem werden die Protokolle SIEMENS/FDC/RCCAR erst ab einer Scan-Frequenz von ca. 15kHz zuverlässig erkannt. Bei LEGO sind es sogar 20kHz. Wenn man also diese Protokolle nutzen will, muss man F_INTERRUPTS entsprechend anpassen, sonst erscheint beim Übersetzen eine entsprechende Warnung und die entsprechenden Protokolle werden dann automatisch abgeschaltet.

IRMP_PORT_LETTER + IRMP_BIT_NUMBER

Über diese Konstanten wird der Pin am µC beschrieben, an welchem der IR-Empfänger angeschlossen ist.

Standardwert ist 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
```

Diese beiden Werte sind an den tatsächlichen Hardware-Pin des µCs anzupassen.

Dies gilt ebenso für die STM32-µCs:

```
/*---------------------------------------------------------------------
 * 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
```

Bei den PIC-Prozessoren gibt es lediglich die anzupassende Konstante **IRMP_PIN** - je nach 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
```

IRMP_USE_CALLBACK

Standardwert:

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

Wenn man Callbacks einschaltet, wird bei jeder Pegeländerung des Eingangs eine Callback-Funktion aufgerufen. Dies kann zum Beispiel dafür verwendet werden, das eingehende IR-Signal sichtbar zu machen, also als Signal an einem weiteren Pin auszugeben.

Hier ein Beispiel:

```
#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 (uint8_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_LOGGING

Mit IRMP_LOGGING kann das Protokollieren von eingehenden IR-Frames eingeschaltet werden.

Standardwert:

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

Weitere Erläuterungen siehe Scannen von unbekannten IR-Protokollen.

Anwendung von IRMP

Die von IRMP unterstützten Protokolle weisen Bitlängen - teilweise variabel, teilweise fest - von 2 bis 48 Bit auf. Diese werden über Preprocessor-Defines beschrieben.

IRMP trennt diese IR-Telegramme prinzipiell in 3 Bereiche:

```
1. ID für verwendetes Protokoll
2. Adresse bzw. Herstellercode
3. Kommando
```

Mittels der Funktion

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

kann man ein decodiertes Telegramm abrufen. Der Return-Wert ist 1, wenn ein Telegramm eingelesen wurde, sonst 0. Im ersten Fall werden die 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)
```

gefüllt.

Das heisst: am Ende bekommt man dann über irmp_get_data() einfach drei Werte (Protokoll, Adresse und Kommando-Code), die man über ein if oder switch checken kann, z. B. hier eine Routine, welche die Tasten 1-9 auf einer Fernbedienung auswertet:

```
  IRMP_DATA irmp_data;

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

Hier die möglichen Werte für irmp_data.protocol, siehe auch irmpprotocols.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpprotocols.h?view=markup):

```
#define IRMP_SIRCS_PROTOCOL                     1               // Sony
#define IRMP_NEC_PROTOCOL                       2               // NEC, Pioneer, JVC, Toshiba, NoName 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 Mediareceiver
```

```
#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)
```

Die Werte für die Adresse und das Kommando muss man natürlich einmal für eine unbekannte Fernbedienung auslesen und dann über ein UART oder LC-Display ausgeben, um sie dann im Programm hart zu kodieren. Oder man hat eine kleine Anlernroutine, wo man einmal die gewünschten Tasten drücken muss, um sie anschließend im EEPROM abzuspeichern. Ein Beispiel dazu findet man im Artikel Lernfähige IR-Fernbedienung mit IRMP (http://www.mikrocontroller.net/articles/DIY_Lernfähige_Fernbedienung_mit_IRMP).

Eine weitere Beispiel-Main-Funktion (http://www.mikrocontroller.net/svnbrowser/irmp/main.c?view=markup) ist im Zip-File enthalten, da sieht man dann auch die Initialisierung des Timers.

"Entprellen" von Tasten

Um zu unterscheiden, ob eine Taste lange gedrückt wurde oder lediglich einzeln, dient das Bit IRMP_FLAG_REPETITION. Dieses wird im Struct-Member **flags** gesetzt, wenn eine Taste auf der Fernbedienung längere Zeit gedrückt wurde und dadurch immer wieder dasselbe Kommando innerhalb kurzer Zeitabstände ausgesandt wird.

Beispiel:

```
if (irmp_data.flags & IRMP_FLAG_REPETITION)
{
  // Benutzer hält die Taste länger runter
  // entweder:
  //   ich ignoriere die (Wiederholungs-)Taste
  // oder:
  //   ich benutze diese Info, um einen Repeat-Effekt zu nutzen
}
else
{
  // Es handelt sich um eine neue Taste
}
```

Dies kann zum Beispiel dafür genutzt werden, um die Tasten 0-9 zu "entprellen", indem man Kommandos mit gesetztem Bit IRMP_FLAG_REPETITION ignoriert. Bei dem Drücken auf die Tasten VOLUME+ oder VOLUME- kann die wiederholte Auswertung ein und desselben Kommandos aber durchaus gewünscht sein - zum Beispiel, um LEDs zu faden.

Wenn man nur Einzeltasten auswerten will, kann man obigen IF-Block reduzieren auf:

```
if (! (irmp_data.flags & IRMP_FLAG_REPETITION))
{
  // Es handelt sich um eine neue Taste
  // ACTION!
}
```

Arbeitsweise

Das "Working Horse" von IRMP ist die Interrupt Service Routine irmp_ISR() welche 15.000 mal pro Sekunde aufgerufen werden sollte. Weicht dieser Wert ab, muss die Preprocessor-Konstante F_INTERRUPTS in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) angepasst werden. Der Wert kann zwischen 10kHz und 20kHz eingestellt werden.

irmp_ISR() detektiert zunächst die Länge und die Form des/der Startbits und ermittelt daraus das verwendete Protokoll. Sobald das Protokoll erkannt wurde, werden die weiter einzulesenden Bits parametrisiert, um dann möglichst effektiv in den weiteren Aufrufen das komplette IR-Telegramm einzulesen.

Um direkt Kritikern den Wind aus den Segeln zu nehmen:

Ich weiss, die ISR ist ziemlich groß. Aber da sie sich wie eine State Machine verhält, ist der tatsächlich ausgeführte Code pro Durchlauf relativ gering. Solange es "dunkel" ist (und das ist es ja die meiste Zeit ;-)) ist die aufgewendete Zeit sogar verschwindend gering. Im WordClock-Projekt werden mit ein- und demselben Timer 8 ISRs aufgerufen, davon ist die irmp_ISR() nur eine unter vielen. Bei mindestens 8 MHz CPU-Takt traten bisher keine Timing-Probleme auf. Daher sehe ich bei der Länge von irmp_ISR überhaupt kein Problem.

Ein Quarz ist nicht unbedingt notwendig, es funktioniert auch mit dem internen Oszillator des AVRs, wenn man die Prescaler-Fuse entsprechend gesetzt hat, dass die CPU auch mit 8MHz rennt ... Die Fuse-Werte für einen ATMEGA88 findet man in main.c (http://www.mikrocontroller.net/svnbrowser/irmp/main.c?view=markup).

Scannen von unbekannten IR-Protokollen

Stellt man in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) in der Zeile

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

den Wert für IRMP_LOGGING auf 1, wird in IRMP eine Protokollierung eingeschaltet: Es werden dann die Hell- und Dunkelphase auf dem UART des Microntrollers mit 9600Bd ausgegeben: 1=Dunkel, 0=Hell. Eventuell müssen dann die Konstanten in den Funktionen uart_init() und uart_putc() angepasst werden; das kommt auf den verwendeten AVR-µC an.

Hinweis: Für PIC-Prozessoren gibt es ein eigenes Logging-Modul namens irmpextlog.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmpextlog.c?view=markup). Dieses ermöglicht das Logging über USB. Für AVR-Prozessoren ist irmpextlog.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmpextlog.c?view=markup) irrelevant

Nimmt man diese Protokoll-Scans mit einem Terminal-Emulationsprogramm auf und speichert sie dann als normale Datei ab, kann man diese Scan-Dateien zur Analyse verwenden, um damit IRMP an das unbekannte Protokoll anzupassen - siehe nächstes Kapitel.

Wer eine Fernbedienung hat, die nicht von IRMP unterstützt wird, kann mir (ukw (http://www.mikrocontroller.net/user/show/ukw)) gern die Scan-Dateien zuschicken. Ich schaue dann, ob das Protokoll in das IRMP-Konzept passt und passe gegebenenfalls den Source an.

IRMP unter Linux und Windows

Übersetzen

irmp.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmp.c?view=markup) lässt sich auch unter Linux direkt kompilieren, um damit Infrarot-Scans, welche in Dateien gespeichert sind, direkt zu testen. Im Unterordner IR-Data finden sich solche Dateien, die man dem IRMP direkt zum "Fraß" vorwerfen kann.

Das Übersetzen von IRMP geht folgendermaßen:

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

Dabei werden 3 IRMP-Versionen erzeugt:

- irmp-10kHz: Version für 10kHz Scans
- irmp-15kHz: Version für 15kHz Scans
- irmp-20kHz: Version für 20kHz Scans

Aufruf von IRMP

Der Aufruf geschieht dann über:

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

Die angegebenen Optionen schließen sich aus, das heisst, es kann jeweils nur eine Option zu einer Zeit angegeben werden:

Option:

```
-l  List            gibt eine Liste der Pulse und Pausen aus
-a analyze          analysiert die Puls-/Pausen und schreibt ein "Spektrum" in ASCII-Form
-v verbose          ausführliche Ausgabe
-p  Print Timings   gibt für alle Protokolle eine Timing-Tabelle aus
```

Beispiele:

Normale Ausgabe

```
./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
-----------------------------------------------------------------------------
...
```

Listen-Ausgabe

```
./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
...
```

Analyse

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

```
-----------------------------------------------------------------------------
START PULSES:
 90 o 1
 91 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 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 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 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 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 562
  7 ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 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 ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 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 ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 425
 17 oooooooooo 72
pause avg: 16.1=1605.4 us, min: 15=1500.0 us, max: 17=1700.0 us, tol:  6.6%
-----------------------------------------------------------------------------
```

Hier sieht man die gemessenen Zeiten aller Pulse und Pausen als (liegende) Glockenkurven, welche natürlich wegen der ASCII-Darstellung nicht gerade einer Idealkurve entsprechen. Je schmaler die gemessenen Kanäle, desto besser ist das Timing der Fernbedienung.

Aus obigem Output kann man herauslesen:

- Das Start-Bit hat eine Pulslänge zwischen 9000 und 9200 usec, im Mittel sind es 9102 usec. Die Abweichung von diesem Mittelwert liegt bei 1,1 Prozent.
- Das Start-Bit hat eine Pausenlänge zwischen 4300 usec und 4500 usec, der Mittelwert beträgt 4424 usec. Der Fehler liegt bei 2,8 Prozent.
- Die Pulslänge eines Datenbits liegt zwischen 500 usec und 700 usec, im Mittel sind es 650 usec, der Fehler liegt bei (stolzen) 23,1 Prozent!

Desweiteren gibt es noch 2 verschieden lange Pausen (für die Bits 0 und 1), das Ablesen der Werte überlasse ich dem geneigten Leser ;-)

Ausführliche Ausgabe

```
./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
-------------------------------------------------------------------------------
```

Aufruf unter Windows

IRMP kann man auch unter Windows nutzen, nämlich folgendermaßen:

- Eingabeaufforderung starten
- In das Verzeichnis irmp wechseln
- Aufruf von:

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

Es gelten dieselben Optionen wie für die Linux-Version.

Längere Ausgaben

Da manche Ausgaben sehr lang werden, empfiehlt es sich auch hier, die Ausgabe in eine Datei zu lenken oder in einen Pager weiterzuleiten, damit man seitenweise blättern kann:

Linux:

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

Windows:

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

Fernbedienungen

Protokoll	Bezeichnung	Gerät	Device Address
NEC	Toshiba CT-9859	Fernseher	0x5F40
	Toshiba VT-728G	V-728G Videorekorder	0x5B44
	Elta 8848 MP 4	DVD-Player	0x7F00
	AS-218	Askey TV-View CHP03X (TV-Karte)	0x3B86
	Cyberhome ???	Cyberhome DVD Player	0x6D72
	WD TV Live	Western Digital Multimediaplayer	0x1F30
	Canon WL-DC100	Kamera Canon PowerShot G5	0xB1CA
NEC16	Daewoo	Videorekorder	0x0015
KASEIKYO	Technics EUR646497	AV Receiver SA-AX 730	0x2002
	Panasonic TV	Fernseher TX-L32EW6	0x2002
RC5	Loewe Assist/RC3/RC4	Fernseher (FB auf TV-Mode)	0x0000
RC6	Philips Television	Fernseher (FB auf TV-Mode)	0x0000
SIRCS	Sony RM-816	Fernseher (FB auf TV-Mode)	0x0000
DENON	DENON RC970	AVR3805 (Verstärker)	0x0008
	DENON RC970	DVD/CD-Player	0x0002
	DENON RC970	Tuner	0x0006
SAMSUNG32	Samsung AA59-00484A	LE40D550 Fernseher	0x0707
	LG AKB72033901	Blu-Ray Player BD370	0x2D2D
APPLE	Apple	Apple Dock (iPod 2)	0x0020

IR-Tastaturen

IRMP unterstützt ab Version 1.7.0 auch IR-Tastaturen, nämlich die Infrarot-Tastatur FDC-3402 - erhältlich bei Pollin (Art. 711 056) für weniger als 2 EUR.


FDC-3402-Tastatur

Beim Erkennen einer Taste gibt IRMP folgende Daten zurück:

```
Protokoll-Nummer (irmp_data.protocol): 18
Addresse         (irmp_data.address):  0x003F
```

Als Kommando (irmp_data.command) werden folgende Werte zurückgeliefert:

Code	Taste	Code	Taste	Code	Taste	Code	Taste	Code	Taste	Code	Taste	Code	Taste	Code	Taste
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	

Zusatztasten links:

Code	Taste
0x0400	KEY_MOUSE_1
0x0800	KEY_MOUSE_2

Dabei gelten die obigen Werte für das Drücken einer Taste. Wird die Taste wieder losgelassen, setzt IRMP zusätzlich das 8. Bit im Kommando.

Beispiel:

```
    Taste 'a' drücken:   0x001F
    Taste 'a' loslassen: 0x009F
```

Ausnahme ist die EIN/AUS-Taste: Diese sendet nur beim Drücken einen Code, nicht beim Loslassen.

Wird eine Taste länger gedrückt, wird das in irmp_data.flag angezeigt.

Beispiel:

```
                      command   flag
   Taste 'a' drücken:   0x001F    0x00
   Taste 'a' drücken:   0x001F    0x01
   Taste 'a' drücken:   0x001F    0x01
   Taste 'a' drücken:   0x001F    0x01
   ....
   Taste 'a' loslassen: 0x009F    0x00
```

Werden Tastenkombinationen (zum Beispiel für ein großes 'A') gedrückt, dann sind die Rückgabewerte von IRMP in folgendem Ablauf zu sehen:

```
   Linke SHIFT-Taste drücken:   0x0002
   Taste 'a' drücken:           0x001F
   Taste 'a' loslassen:         0x009F
   Linke SHIFT-Taste loslassen: 0x0082
```

In irmp.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmp.c?view=markup) findet man für die LINUX-Version eine Funktion get_fdc_key(), welche als Vorlage dienen mag, die Keycodes einer FDC-Tastatur in die entsprechenden ASCII-Codes umzuwandeln. Diese Funktion kann man entweder lokal auf dem µC nutzen, um die Keycodes zu decodieren, oder auf einem Hostsystem (z.B. PC), an welches die IRMP-Data-Struktur gesandt wird. Dafür sollte man die Funktion incl. der dazugehörenden Preprozessor-Konstanten in seinen Applikations-Quelltext kopieren.

Hier der entsprechende Auszug:

```
#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', 'ß', '´',  0,  '\b',
       '\t','q', 'w', 'e', 'r', 't', 'z', 'u', 'i', 'o', 'p', 'ü', '+',  0,   0,  'a',
       's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'ö', 'ä', '#',  '\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', 'Ü', '*',  0,   0,  'A',
       'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'Ö', 'Ä', '\'','\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 = '²';     break;
                            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 = 'µ';     break;
                        }
                    }
                    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);
}
```

Als letztes noch ein Beispiel einer Anwendung der Funktion get_fdc_key():

```
    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);
            }
        }
    }
```

Alle nicht-druckbaren Zeichen werden dabei folgendermaßen codiert:

Taste	Konstante	Wert
ESC	KEY_ESCAPE	0x1B
Menü	KEY_MENUE	0x80
Zurück	KEY_BACK	0x81
Vorw.	KEY_FORWARD	0x82
Adresse	KEY_ADDRESS	0x83
Fenster	KEY_WINDOW	0x84
1. Seite	KEY_1ST_PAGE	0x85
Stop	KEY_STOP	0x86
Mail	KEY_MAIL	0x87

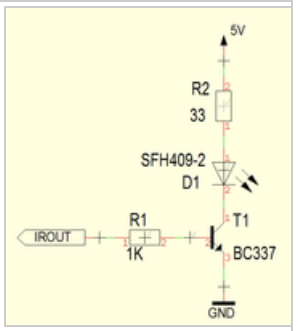
Fav.	KEY_FAVORITES	0x88
Neue Seite	KEY_NEW_PAGE	0x89
Setup	KEY_SETUP	0x8A
Schrift	KEY_FONT	0x8B
Druck	KEY_PRINT	0x8C
Ein/Aus	KEY_ON_OFF	0x8E
Backspace	'\b'	0x08
CR/ENTER	'\r'	0x0C
TAB	'\t'	0x09
Einfg	KEY_INSERT	0x90
Entf	KEY_DELETE	0x91
Cursor links	KEY_LEFT	0x92
Pos1	KEY_HOME	0x93
Ende	KEY_END	0x94
Cursor rechts	KEY_UP	0x95
Cursor runter	KEY_DOWN	0x96
Bild hoch	KEY_PAGE_UP	0x97
Bild runter	KEY_PAGE_DOWN	0x98
Cursor links	KEY_RIGHT	0x99
Linke Maustaste	KEY_MOUSE_1	0x9E
Rechte Maustaste	KEY_MOUSE_2	0x9F

Die Funktion get_fdc_key berücksichtigt das Gedrückthalten der Shift-, Strg- und ALT-Tasten. Damit funktioniert nicht nur das Schreiben von Großbuchstaben, sondern auch das Auswählen der Sonderzeichen mit der Tastenkombination ALT + Taste, z.B. ALT + m = µ oder ALT + q = @. Ebenso kann man mit der Strg-Taste die Control-Zeichen CTRL-A bis CTRL-Z senden. Die CapsLock-Taste wird ignoriert, da ich sie sowieso für die überflüssigste Taste überhaupt halte ;-)

IRSND - Infrarot-Multiprotokoll-Encoder

Einleitung IRSND

IRSND ist das Gegenstück zu IRMP: es reproduziert aus den Daten, die mit IRMP empfangen wurden, wieder den Original Frame, der dann über eine Infrarot-Diode ausgegeben werden kann.



Anschluß eines einfachen IR-Senders an µC.

Von IRSND unterstützte µCs

IRSND ist lauffähig auf folgenden AVR µCs:

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

Es gibt aber auch Portierungen auf diverse PIC µCs - für den CCS- und C18-Compiler. Auch ist IRSND mittlerweile auf ARM STM32 lauffähig.

Von IRSND unterstützte Protokolle

IRSND unterstützt die folgenden Protokolle:

- SIRCS
- NEC
- SAMSUNG
- SAMSUNG32
- SAMSUNG48
- MATSUSHITA
- RC5
- KASEIKYO
- DENON

- JVC
- APPLE
- NUBERT
- BANG_OLUFSON
- GRUNDIG
- NOKIA
- SIEMENS (bei mind. ~15kHz)
- FDC (bei mind. ~15kHz)
- RCCAR (bei mind. ~15kHz)
- RECS80 (bei mind. ~20kHz)
- RECS80EXT (bei mind. ~20kHz)
- NIKON
- RC6
- RC6A
- THOMSON
- NEC16
- NEC42
- LEGO
- IR60 (SDA2008)
- A1TVBOX
- ROOMBA **(NEU!)**
- SPEAKER **(NEU!)**
- TELEFUNKEN **(NEU!)**

IRSND unterstützt die folgenden Protokolle derzeit (noch) **NICHT**:

- KATHREIN
- NETBOX
- ORTEK
- RCMM
- LGAIR

Download IRSND

Version 2.6.4, Stand vom 15.09.2014

Download Release-Version: Irsnd.zip (http://www.mikrocontroller.net/wikifiles/c/c7/Irsnd.zip)

IRMP & IRSND sind nun auch über SVN abrufbar: IRMP im SVN (http://www.mikrocontroller.net/svnbrowser/irmp/)

Achtung:

Die Version im SVN kann eine Zwischen- oder Test-Version sein, die nicht den hier dokumentierten Stand widerspiegelt! Im Zweifel verwendet man besser den obigen Download auf Irsnd.zip.

Die Software-Änderungen kann man sich hier anschauen: Software-Historie IRSND (http://www.mikrocontroller.net/articles/IRMP#Software-Historie_IRSND)

Source-Code IRSND

Der Source-Code lässt sich einfach übersetzen, indem man unter Windows die Projekt-Datei irsnd.aps in das AVRStudio 4 lädt.

Auch für andere Entwicklungsumgebungen lässt sich leicht ein Projekt bzw. Makefile zusammenstellen. Zum IRSND-Source gehören folgende Dateien:

- irsnd.c (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.c?view=markup) - Der eigentliche IR-Encoder
- irmpprotocols.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpprotocols.h?view=markup) - Sämtliche Definitionen zu den IR-Protokollen
- irmpsystem.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpsystem.h?view=markup) - Vom Zielsystem abhängige Definitionen für AVR/PIC/STM32
- irsnd.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.h?view=markup) - Include-Datei für die Applikation
- irsndconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsndconfig.h?view=markup) - Anzupassende Konfigurationsdatei
- irsndmain.c (http://www.mikrocontroller.net/svnbrowser/irmp/irsndmain.c?view=markup) - Beispiel Anwendung

WICHTIG:

Im Applikations-Source sollte nur irsnd.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.h?view=markup) per include eingefügt werden, also lediglich:

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

Alle anderen Include-Dateien werden automatisch über irsnd.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.h?view=markup) "eingefügt". Siehe dazu auch die Beispieldatei irsndmain.c (http://www.mikrocontroller.net/svnbrowser/irmp/irsndmain.c?view=markup).

IRSND encodiert sämtliche oben aufgelisteten Protokolle in einer ISR, siehe irsnd.c (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.c?view=markup).

Einstellungen in irsndconfig.h

F_INTERRUPTS

Anzahl der Interrupts pro Sekunde. Der Wert kann zwischen 10000 und 20000 eingestellt werden.

Standardwert:

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

IRSND_SUPPORT_xxx_PROTOCOL

Hier lässt sich einstellen, welche Protokolle von IRSND unterstützt werden sollen. Die Standardprotokolle sind bereits aktiv. Möchte man weitere Protokolle einschalten bzw. einige aus Speicherplatzgründen deaktivieren, sind die entsprechenden Werte in irsndconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsndconfig.h?view=markup) anzupassen.

```
// typical protocols, disable here!             Enable  Remarks                 F_INTERRUPTS            Program Space
#define IRSND_SUPPORT_SIRCS_PROTOCOL            1       // Sony SIRCS           >= 10000                ~200 bytes
#define IRSND_SUPPORT_NEC_PROTOCOL              1       // NEC + APPLE          >= 10000                ~100 bytes
#define IRSND_SUPPORT_SAMSUNG_PROTOCOL          1       // Samsung + Samsung32  >= 10000                ~300 bytes
#define IRSND_SUPPORT_MATSUSHITA_PROTOCOL       1       // Matsushita           >= 10000                ~200 bytes
#define IRSND_SUPPORT_KASEIKYO_PROTOCOL         1       // Kaseikyo             >= 10000                ~300 bytes

// more protocols, enable here!                 Enable  Remarks                 F_INTERRUPTS            Program Space
#define IRSND_SUPPORT_DENON_PROTOCOL            0       // DENON, Sharp         >= 10000                ~200 bytes
#define IRSND_SUPPORT_RC5_PROTOCOL              0       // RC5                  >= 10000                ~150 bytes
#define IRSND_SUPPORT_RC6_PROTOCOL              0       // RC6                  >= 10000                ~250 bytes
#define IRSND_SUPPORT_RC6A_PROTOCOL             0       // RC6A                 >= 10000                ~250 bytes
#define IRSND_SUPPORT_JVC_PROTOCOL              0       // JVC                  >= 10000                ~150 bytes
#define IRSND_SUPPORT_NEC16_PROTOCOL            0       // NEC16                >= 10000                ~150 bytes
#define IRSND_SUPPORT_NEC42_PROTOCOL            0       // NEC42                >= 10000                ~150 bytes
#define IRSND_SUPPORT_IR60_PROTOCOL             0       // IR60 (SDA2008)       >= 10000                ~250 bytes
#define IRSND_SUPPORT_GRUNDIG_PROTOCOL          0       // Grundig              >= 10000                ~300 bytes
#define IRSND_SUPPORT_SIEMENS_PROTOCOL          0       // Siemens, Gigaset     >= 15000                ~150 bytes
#define IRSND_SUPPORT_NOKIA_PROTOCOL            0       // Nokia                >= 10000                ~400 bytes

// exotic protocols, enable here!               Enable  Remarks                 F_INTERRUPTS            Program Space
#define IRSND_SUPPORT_KATHREIN_PROTOCOL         0       // Kathrein             >= 10000                DON'T CHANGE, NOT SUPPORTED YET!
#define IRSND_SUPPORT_NUBERT_PROTOCOL           0       // NUBERT               >= 10000                ~100 bytes
#define IRSND_SUPPORT_BANG_OLUFSEN_PROTOCOL     0       // Bang&Olufsen         >= 10000                ~250 bytes
#define IRSND_SUPPORT_RECS80_PROTOCOL           0       // RECS80               >= 15000                ~100 bytes
#define IRSND_SUPPORT_RECS80EXT_PROTOCOL        0       // RECS80EXT            >= 15000                ~100 bytes
#define IRSND_SUPPORT_THOMSON_PROTOCOL          0       // Thomson              >= 10000                ~250 bytes
#define IRSND_SUPPORT_NIKON_PROTOCOL            0       // NIKON                >= 10000                ~150 bytes
#define IRSND_SUPPORT_NETBOX_PROTOCOL           0       // Netbox keyboard      >= 10000                DON'T CHANGE, NOT SUPPORTED YET!
#define IRSND_SUPPORT_FDC_PROTOCOL              0       // FDC IR keyboard      >= 10000 (better 15000)  ~150 bytes
#define IRSND_SUPPORT_RCCAR_PROTOCOL            0       // RC CAR               >= 10000 (better 15000)  ~150 bytes
#define IRSND_SUPPORT_ROOMBA_PROTOCOL           0       // iRobot Roomba        >= 10000                ~150 bytes
#define IRSND_SUPPORT_RUWIDO_PROTOCOL           0       // RUWIDO, T-Home       >= 15000                ~250 bytes
#define IRSND_SUPPORT_A1TVBOX_PROTOCOL          0       // A1 TV BOX            >= 15000 (better 20000)  ~200 bytes
#define IRSND_SUPPORT_LEGO_PROTOCOL             0       // LEGO Power RC        >= 20000                ~150 bytes
```

Mit Setzen auf 0 wird das Protokoll deaktiviert, mit Setzen auf 1 wird es aktiviert. Die deaktivierten Protokolle werden dann nicht mitübersetzt. Das spart Speicherplatz im Flash, siehe Angaben in obigen Kommentaren. Wenn man unbedingt Speicherplatz sparen muss, gelten natürlich hier dieselben Tipps wie für IRMP.

Um das APPLE-Protokoll zu unterstützen, ist IRSND_SUPPORT_NEC_PROTOCOL auf 1 zu setzen, da es sich hier lediglich um einen Spezialfall vom NEC-Protokoll handelt.

IRSND_OCx

Für das Senden der IR-Signale benötigt IRSND einen PWM-fähigen Output-Pin, da das Signal moduliert werden muss. Möglich sind eine der folgenden Einstellungen:

```
#define IRSND_OCx                               IRSND_OC2  // OC2  on ATmegas         supporting OC2,  e.g. ATmega8
#define IRSND_OCx                               IRSND_OC2A // OC2A on ATmegas         supporting OC2A, e.g. ATmega88
#define IRSND_OCx                               IRSND_OC2B // OC2B on ATmegas         supporting OC2B, e.g. ATmega88
#define IRSND_OCx                               IRSND_OC0  // OC0  on ATmegas         supporting OC0,  e.g. ATmega162
#define IRSND_OCx                               IRSND_OC0A // OC0A on ATmegas/ATtinys supporting OC0A, e.g. ATtiny84, ATtiny85
#define IRSND_OCx                               IRSND_OC0B // OC0B on ATmegas/ATtinys supporting OC0B, e.g. ATtiny84, ATtiny85
```

Standardwert:

```
#define IRSND_OCx                               IRSND_OC2B
```

Für die PIC- und STM32-µCs sind entsprechende Werte anzupassen, siehe Kommentare in irsndconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsndconfig.h?view=markup).

IRSND_USE_CALLBACK

Standardwert:

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

Wenn man Callbacks einschaltet, wird bei jeder Änderung des Signals (IR-Modulation ein/aus) eine Callback-Funktion aufgerufen. Dies kann zum Beispiel dafür verwendet werden, ein unmoduliertes Signal an einem weiteren Pin auszugeben.

Hier ein Beispiel:

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

/*---------------------------------------------------------------------------------------------------------------
 * Called (back) from IRSND module
 * This example switches a LED (which is connected to Vcc)
 *---------------------------------------------------------------------------------------------------------------
 */
void
led_callback (uint8_t on)
{
    if (on)
    {
        LED_PORT &= ~(1 << LED_PIN);
    }
    else
    {
        LED_PORT |= (1 << LED_PIN);
    }
}

int
main ()
{
    ...
    LED_DDR |= (1 << LED_PIN);          // LED pin to output
    LED_PORT |= (1 << LED_PIN);         // switch LED off (active low)
    irsnd_init ();
    irsnd_set_callback_ptr (led_callback);
    sei ();
    ...
}
```

Anwendung von IRSND

IRSND baut den zu sendenden Frame "on-the-fly" aus der IRMP-Datenstruktur wieder zusammen. Dazu zählen:

```
1. ID für verwendetes Protokoll
2. Adresse bzw. Herstellercode
3. Kommando
```

Mittels der Funktion

```
 irsnd_send_data (IRMP_DATA * irmp_data_p)
```

kann man ein zu encodierendes Telegramm versenden. Der Return-Wert ist 1, wenn das Telegramm versendet werden kann, sonst 0. Im ersten Fall werden die Struct-Members

```
    irmp_data_p->protocol
    irmp_data_p->address
    irmp_data_p->command
    irmp_data_p->flags
```

ausgelesen und dann als Frame im jeweils gewünschten Protokoll gesendet.

irmp_data_p->flags gibt die Anzahl der Wiederholungen an, z.B.

```
irmp_data_p->flags = 0: keine Wiederholung
irmp_data_p->flags = 1: 1 Wiederholung
irmp_data_p->flags = 2: 2 Wiederholungen
usw.
```

Zu beachten: Es ist unbedingt darauf zu achten, dass irmp_data_p->flags vor dem Aufruf von irsnd_send_data() einen definierten Wert hat!

Hier ein Beispiel:

```
  IRMP_DATA irmp_data;

  irmp_data.protocol = IRMP_NEC_PROTOCOL;       // sende im NEC-Protokoll
  irmp_data.address  = 0x00FF;                  // verwende Adresse 0x00FF
  irmp_data.command  = 0x0001;                  // sende Kommando 0001
  irmp_data.flags    = 0;                       // keine Wiederholung!

  (void) irsnd_send_data (&irmp_data, FALSE);   // versende ohne Prüfung und ohne Warten
```

Der Frame wird asynchron über die Interrupt-Routine irsnd_ISR() verschickt, so dass die Funktion irsnd_send_data() sofort zurückkommt.

Sind Wiederholungen angegeben, wird entweder der Frame nach einer Pause (protokollabhängig) neu ausgegeben oder ein protokollspezifischer Wiederholungsframe (z.B. für NEC) gesendet.

Wird erneut irsnd_send_data() aufgerufen, wartet diese, bis der vorhergenhende Frame vollständig verschickt wurde. Man kann aber auch selbst prüfen, ob IRSND gerade "busy" ist oder nicht:

```
  while (irsnd_is_busy ())
  {
    ;                                           // selber warten oder was anderes tun...
  }
  (void) irsnd_send_data (&irmp_data, FALSE);   // versende ohne Prüfung und ohne Warten
```

Wird irsnd_send_data() mit dem 2. Argument TRUE aufgerufen, kommt diese Funktion erst zurück, wenn der Frame komplett ausgesendet wurde.

Im Beispiel-Source irsndmain.c findet man neben der Verwendung von irsnd_send_data() auch den Timer-Aufruf:

```
ISR(TIMER1_COMPA_vect)
{
  irsnd_ISR())          // call irsnd ISR
  // call other timer interrupt routines...
}
```

Paralleles Betreiben von IRMP und IRSND

Möchte man IRMP und IRSND parallel verwenden (also als Sender und Empfänger) schreibt man die ISR folgendermaßen:

```
ISR(TIMER1_COMPA_vect)
{
  if (! irsnd_ISR())          // call irsnd ISR
  {                           // if not busy...
      irmp_ISR();             // call irmp ISR
  }
  // call other timer interrupt routines...
}
```

Das heisst: Nur wenn irsnd_ISR() nichts zu tun hat, dann rufe die ISR des Empfängers auf. Damit ist der Empfänger solange abgeschaltet, während irsnd_ISR() noch Daten sendet. Die Timer-Initialisierungsroutine ist für IRMP und IRSND dann natürlich dieselbe.

Eine gemeinsame main-Funktion könnte dann zum Beispiel folgendermaßen aussehen:

```
int
main (void)
{
  IRMP_DATA irmp_data;

  irmp_init();                // initialize irmp
  irsnd_init();               // initialize irsnd
  timer_init();               // initialize timer
  sei ();                     // enable interrupts

  for (;;)
  {
    if (irmp_get_data (&irmp_data))
    {
      irmp_data.flags = 0;    // reset flags!
      irsnd_send_data (&irmp_data);
    }
  }
}
```

Die Funktion des obigen Sources ist offensichtlich: Ein empfangenes Telegramm wird nach vollständiger Decodierung wieder encodiert und dann wieder über die IR-Diode ausgesandt. Somit können dann zum Beispiel Signale "um die Ecke" oder streckenweise drahtgebunden übertragen werden.

Desweiteren könnte man auch Protokolle transformieren, zum Beispiel NEC-Telegramme in RC5 umwandeln, wenn man seine Original-RC5-FB zu seinem Philips-Gerät verlegt hat...

Der Rest bleibt der Phantasie des geneigten Lesers überlassen ;-)

Hier noch die möglichen Werte für irmp_data.protocol, siehe auch irmpprotocols.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpprotocols.h?view=markup):

```
#define IRMP_SIRCS_PROTOCOL                     1           // Sony
#define IRMP_NEC_PROTOCOL                       2           // NEC, Pioneer, JVC, Toshiba, NoName 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 Mediareceiver
#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)
```

Die Daten für die Adresse und das Kommando ermittelt man am besten über IRMP, siehe weiter oben ;-)

IRSND unter Linux und Windows

Übersetzen von IRSND

irsnd.c (http://www.mikrocontroller.net/svnbrowser/irmp/irsnd.c?view=markup) lässt sich auch unter Linux direkt kompilieren, um damit Telegramme in Form von IRMP-Scan-Dateien zu erzeugen. Das geht folgendermaßen:

```
make -f makefile.unx
```

Aufruf von IRSND

Der Aufruf geht dann folgendermaßen:

```
./irsnd protocol-number hex-address hex-command [repeat] > filename.txt
```

also zum Beispiel für das NEC-Protokoll, Adresse 0x00FF, Kommando 0x0001

```
./irsnd 2 00FF 0001 > nec.txt                  # irsnd ausführen
```

IRSND unter Windows

IRSND kann man auch unter Windows nutzen, nämlich folgendermaßen:

- Eingabeaufforderung starten
- In das Verzeichnis von irsnd wechseln
- Aufruf von:

```
        irsnd.exe 2 00FF 0001 > nec.txt
```

Nun kann man direkt mit IRMP anschließend testen, ob das erzeugte Telegramm auch korrekt ist:

```
        ./irmp < nec.txt
```

bzw. unter Windows:

```
        irmp.exe < nec.txt
```

Das Ganze geht auch ohne Zwischendatei, nämlich:

```
        ./irsnd 2 00FF 0001 | ./irmp
```

bzw. unter Windows:

```
        irsnd.exe 2 00FF 0001 | irmp.exe
```

IRMP gibt dann als Ergebnis folgendes aus:

```
        11111111000000001000000001111111 p =  2, a = 0x00ff, c = 0x0001, f = 0x00
```

IRMP konnte also aus dem von IRSND generierten Frame wieder das Protokoll 2, Adresse 0x00FF und Kommando 0x0001 decodieren.

Bitte beachten: Je nach benutztem Protokoll sind die Bit-Breiten der Adressen bzw. Kommandos verschieden, siehe obige Tabelle [1] (http://www.mikrocontroller.net/articles/IRMP#Protokolle).

Man kann also nicht mit jedem IR-Protokoll komplett 16-Bit breite Adressen oder Kommandos transparent übertragen.

Anhang

Die IR-Protokolle im Detail

Pulse Distance Protokolle


Pulse Distance Coding

NEC + extended NEC

NEC + extended NEC	Wert
Frequenz	36 kHz / 38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 32 Daten-Bits + 1 Stop-Bit
Daten NEC	8 Adress-Bits + 8 invertierte Adress-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Daten ext. NEC	16 Adress-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Start-Bit	9000µs Puls, 4500µs Pause
0-Bit	560µs Puls, 560µs Pause
1-Bit	560µs Puls, 1690µs Pause
Stop-Bit	560µs Puls
Wiederholung	keine
Tasten-Wiederholung	9000µs Puls, 2250µs Pause, 560µs Puls, ~100ms Pause
Bit-Order	LSB first

JVC

JVC	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 16 Daten-Bits + 1 Stop-Bit
Daten	4 Adress-Bits + 12 Kommando-Bits
Start-Bit	9000µs Puls, 4500µs Pause, 6000µs Pause bei Tasten-Wiederholung
0-Bit	560µs Puls, 560µs Pause
1-Bit	560µs Puls, 1690µs Pause
Stop-Bit	560µs Puls
Wiederholung	keine
Tasten-Wiederholung	Wiederholung nach Pause von 25ms
Bit-Order	LSB first

NEC16

NEC16	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 8 Adress-Bits + 1 Sync-Bit + 8 Daten-Bits + 1 Stop-Bit
Start-Bit	9000µs Puls, 4500µs Pause
Sync-Bit	560µs Puls, 4500µs Pause
0-Bit	560µs Puls, 560µs Pause
1-Bit	560µs Puls, 1690µs Pause

Stop-Bit	560µs Puls
Wiederholung	keine/eine/zwei nach 25ms?
Tasten-Wiederholung	Wiederholung nach Pause von 25ms?
Bit-Order	LSB first

NEC42

NEC42	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 42 Daten-Bits + 1 Stop-Bit
Daten	13 Adress-Bits + 13 invertierte Adress-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Start-Bit	9000µs Puls, 4500µs Pause
0-Bit	560µs Puls, 560µs Pause
1-Bit	560µs Puls, 1690µs Pause
Stop-Bit	560µs Puls
Wiederholung	keine
Tasten-Wiederholung	nach 110ms (ab Start-Bit), 9000µs Puls, 2250µs Pause, 560µs Puls
Bit-Order	LSB first

LGAIR

LGAIR	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 28 Daten-Bits + 1 Stop-Bit
Daten	8 Adress-Bits + 16 Kommando-Bits + 4 Checksum-Bits
Start-Bit	9000µs Puls, 4500µs Pause (identisch mit NEC)
0-Bit	560µs Puls, 560µs Pause (identisch mit NEC)
1-Bit	560µs Puls, 1690µs Pause (identisch mit NEC)
Stop-Bit	560µs Puls (identisch mit NEC)
Wiederholung	keine
Tasten-Wiederholung	unbekannt
Bit-Order	MSB first (**abweichend** zu NEC)

SAMSUNG

SAMSUNG	Wert
Frequenz	?? kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 16 Daten(1)-Bits + 1 Sync-Bit + 20 Daten(2)-Bits + 1 Stop-Bit
Daten(1)	16 Adress-Bits
Daten(2)	4 ID-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Start-Bit	4500µs Puls, 4500µs Pause
0-Bit	550µs Puls, 550µs Pause
1-Bit	550µs Puls, 1650µs Pause
Sync-Bit	550µs Puls, 4500µs Pause
Stop-Bit	550µs Puls
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	LSB first

SAMSUNG32

SAMSUNG32	Wert

Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 32 Daten-Bits + 1 Stop-Bit
Daten	16 Adress-Bits + 16 Kommando-Bits
Start-Bit	4500µs Puls, 4500µs Pause
0-Bit	550µs Puls, 550µs Pause
1-Bit	550µs Puls, 1650µs Pause
Stop-Bit	550µs Puls
Wiederholung	eine nach ca. 47 msec
Tasten-Wiederholung	dritter, fünfter, siebter usw. identischer Frame
Bit-Order	LSB first

SAMSUNG48

SAMSUNG48	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 48 Daten-Bits + 1 Stop-Bit
Daten	16 Adress-Bits + 32 Kommando-Bits
Kommando	8 Bits + 8 invertierte Bits + 8 Bits + 8 invertierte Bits
Start-Bit	4500µs Puls, 4500µs Pause
0-Bit	550µs Puls, 550µs Pause
1-Bit	550µs Puls, 1650µs Pause
Stop-Bit	550µs Puls
Wiederholung	eine nach ca. 5 msec
Tasten-Wiederholung	dritter, fünfter, siebter usw. identischer Frame
Bit-Order	LSB first

MATSUSHITA

MATSUSHITA	Wert
Frequenz	36 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 24 Daten-Bits + 1 Stop-Bit
Daten	6 Hersteller-Bits + 6 Kommando-Bits + 12 Adress-Bits
Start-Bit	3488µs Puls, 3488µs Pause
0-Bit	872µs Puls, 872µs Pause
1-Bit	872µs Puls, 2616µs Pause
Stop-Bit	872µs Puls
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames nach 40ms Pause
Bit-Order	LSB first?

KASEIKYO

KASEIKYO	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 48 Daten-Bits + 1 Stop-Bit
Daten	16 Hersteller-Bits + 4 Parity-Bits + 4 Genre1-Bits + 4 Genre2-Bits + 10 Kommando-Bits + 2 ID-Bits + 8 Parity-Bits
Start-Bit	3380µs Puls, 1690µs Pause
0-Bit	423µs Puls, 423µs Pause
1-Bit	423µs Puls, 1269µs Pause
Stop-Bit	423µs Puls
Wiederholung	einmalig nach 74ms Pause

Tasten-Wiederholung	N-fache Wiederholung des 1. Original-Frames nach ca. 80ms Pause
Bit-Order	LSB first?

RECS80

RECS80	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bits + 10 Daten-Bits + 1 Stop-Bit
Daten	1 Toggle-Bit + 3 Adress-Bits + 6 Kommando-Bits
Start-Bit	158µs Puls, 7432µs Pause
0-Bit	158µs Puls, 4902µs Pause
1-Bit	158µs Puls, 7432µs Pause
Stop-Bit	158µs Puls
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	MSB first

RECS80EXT

RECS80EXT	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	2 Start-Bits + 11 Daten-Bits + 1 Stop-Bit
Daten	1 Toggle-Bit + 4 Adress-Bits + 6 Kommando-Bits
Start-Bit	158µs Puls, 3637µs Pause
0-Bit	158µs Puls, 4902µs Pause
1-Bit	158µs Puls, 7432µs Pause
Stop-Bit	158µs Puls
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	MSB first

DENON

DENON	Wert
Frequenz	38 kHz (in der Praxis, lt. Dokumentation: 32 kHz)
Kodierung	Pulse Distance
Frame	0 Start-Bits + 15 Daten-Bits + 1 Stop-Bit
Daten	5 Address-Bits + 10 Kommando-Bits
Kommando	6 Datenbits + 2 Extension Bits + 2 Data Construction Bits (normal: 00, invertiert: 11)
Start-Bit	kein Start-Bit
0-Bit	310µs Puls, 745µs Pause (in der Praxis, lt. Doku: 275µs Puls, 775µs Pause)
1-Bit	310µs Puls, 1780µs Pause (in der Praxis, lt. Doku: 275µs Puls, 1900µs Pause)
Stop-Bit	310µs Puls (310µs Puls, 745µs Pause (in der Praxis, lt. Doku: 275µs Puls)
Wiederholung	Nach 65ms Wiederholung des Frames mit invertieren Kommando-Bits (Data Construction Bits = 11)
Tasten-Wiederholung	N-fache Wiederholung der beiden Original-Frames nach 65ms
Bit-Order	MSB first

APPLE

APPLE	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 32 Daten-Bits + 1 Stop-Bit

Daten	16 Adress-Bits + 11100000 + 8 Kommando-Bits
Start-Bit	siehe NEC
0-Bit	siehe NEC
1-Bit	siehe NEC
Stop-Bit	siehe NEC
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	LSB first

BOSE

BOSE	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 16 Daten-Bits + 1 Stop-Bit
Daten	0 Adress-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Start-Bit	1060µs Puls, 1425µs Pause
0-Bit	550µs Puls, 437µs Pause
1-Bit	550µs Puls, 1425µs Pause
Stop-Bit	550µs Puls
Wiederholung	keine
Tasten-Wiederholung	noch ungeklärt
Bit-Order	LSB first

B&O

B&O	Wert
Frequenz	455 kHz
Kodierung	Pulse Distance
Frame	4 Start-Bits + 16 Daten-Bits + 1 Trailer-Bit + 1 Stop-Bit
Daten	0 Adress-Bits + 16 Kommando-Bits
Start-Bit 1	200µs Puls, 2925µs Pause
Start-Bit 2	200µs Puls, 2925µs Pause
Start-Bit 3	200µs Puls, 15425µs Pause
Start-Bit 4	200µs Puls, 2925µs Pause
0-Bit	200µs Puls, 2925µs Pause
1-Bit	200µs Puls, 9175µs Pause
R-Bit	200µs Puls, 6050µs Pause, wiederholt das letzte Bit (repetition)
Trailer-Bit	200µs Puls, 12300µs Pause
Stop-Bit	200µs Puls
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	MSB first

FDC

FDC	Wert
Frequenz	38 kHz
Kodierung	Pulse Distance
Frame	1 Start-Bit + 40 Daten-Bits + 1 Stop-Bit
Daten	8 Adress-Bits + 12 x 0-Bits + 4 Press/Release-Bits + 8 Kommando-Bits + 8 invertierte Kommando-Bits
Start-Bit	2085µs Puls, 966µs Pause
0-Bit	300µs Puls, 220µs Pause
1-Bit	300µs Puls, 715µs Pause
Stop-Bit	300µs Puls

Wiederholung	keine
Tasten-Drücken	Press/Release-Bits = 0000
Tasten-Loslassen	Press/Release-Bits = 1111
Tasten-Wiederholung	Wiederholung nach Pause von 60ms
Bit-Order	LSB first

NIKON

NIKON	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 2 Daten-Bits + 1 Stop-Bit
Daten	2 Kommando-Bits
Start-Bit	2200µs Puls, 27100µs Pause
0-Bit	500µs Puls, 1500µs Pause
1-Bit	500µs Puls, 3500µs Pause
Stop-Bit	500µs Puls
Wiederholung	keine
Tasten-Wiederholung	unbekannt
Bit-Order	MSB first

KATHREIN

KATHREIN	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 11 Daten-Bits + 1 Stop-Bit
Daten	4 Adress-Bits + 7 Kommando-Bits
Start-Bit	210µs Puls, 6218µs Pause
0-Bit	210µs Puls, 1400µs Pause
1-Bit	210µs Puls, 3000µs Pause
Stop-Bit	210µs Puls
Wiederholung	keine
Tasten-Wiederholung	nach 35ms?
Bit-Order	MSB first

LEGO

LEGO	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 16 Daten-Bits + 1 Stop-Bit
Daten	16 Kommando-Bits
Start-Bit	158µs Puls, 1026µs Pause
0-Bit	158µs Puls, 263µs Pause
1-Bit	158µs Puls, 553µs Pause
Stop-Bit	158µs Puls
Wiederholung	keine
Tasten-Wiederholung	unbekannt
Bit-Order	MSB first

THOMSON

THOMSON	Wert
Frequenz	33 kHz

Kodierung	Pulse Distance
Frame	0 Start-Bits + 12 Daten-Bits + 1 Stop-Bit
Daten	4 Adress-Bits + 1 Toggle-Bit + 7 Kommando-Bits
0-Bit	550µs Puls, 2000µs Pause
1-Bit	550µs Puls, 4500µs Pause
Stop-Bit	550µs Puls
Wiederholung	keine
Tasten-Wiederholung	Framewiederholung nach 35ms
Bit-Order	vermutlich MSB first

TELEFUNKEN

TELEFUNKEN	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 15 Daten-Bits + 1 Stop-Bit
Daten	0 Adress-Bits + 15 Kommando-Bits
Start-Bit	600µs Puls, 1500µs Pause
0-Bit	600µs Puls, 600µs Pause
1-Bit	600µs Puls, 1500µs Pause
Stop-Bit	600µs Puls
Wiederholung	keine
Tasten-Wiederholung	unbekannt
Bit-Order	vermutlich MSB first

RCCAR

RCCAR	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance
Frame	1 Start-Bit + 13 Daten-Bits + 1 Stop-Bit
Daten	13 Kommando-Bits
Start-Bit	2000µs Puls, 2000µs Pause
0-Bit	600µs Puls, 900µs Pause
1-Bit	600µs Puls, 450µs Pause
Stop-Bit	600µs Puls
Wiederholung	keine
Tasten-Wiederholung	nach 40ms?
Bit-Order	LSB first

RCMM

RCMM	Wert
Frequenz	36 kHz
Kodierung	Pulse Distance
Frame RCMM32	1 Start-Bit + 32 Daten-Bits + 1 Stop-Bit
Frame RCMM24	1 Start-Bit + 24 Daten-Bits + 1 Stop-Bit
Frame RCMM12	1 Start-Bit + 12 Daten-Bits + 1 Stop-Bit
Daten RCMM32	16 Adress-Bits (= 4 Mode-Bits + 12 Device-Bits) + 1 Toggle-Bit + 15 Kommando-Bits
Daten RCMM24	16 Adress-Bits (= 4 Mode-Bits + 12 Device-Bits) + 1 Toggle-Bit + 7 Kommando-Bits
Daten RCMM12	4 Adress-Bits (= 2 Mode-Bits + 2 Device-Bits) + 8 Kommando-Bits
Start-Bit	500µs Puls, 220µs Pause
00-Bits	230µs Puls, 220µs Pause
01-Bits	230µs Puls, 380µs Pause

10-Bits	230µs Puls, 550µs Pause
11-Bits	230µs Puls, 720µs Pause
Stop-Bit	230µs Puls
Wiederholung	keine
Tasten-Wiederholung	nach 80ms
Bit-Order	LSB first

Pulse Width Protokolle


Pulse Width Coding

SIRCS

SIRCS	Wert
Frequenz	40 kHz
Kodierung	Pulse Width
Frame	1 Start-Bit + 12-20 Daten-Bits, kein Stop-Bit
Daten	7 Kommando-Bits + 5 Adress-Bits + bis zu 8 zusätzliche Bits
Start-Bit	2400µs Puls, 600µs Pause
0-Bit	600µs Puls, 600µs Pause
1-Bit	1200µs Puls, 600µs Pause
Wiederholung	zweimalig nach ca. 25ms, d.h. 2. und 3. Frame
Tasten-Wiederholung	ab dem 4. identischen Frame, Abstand ca. 25ms
Bit-Order	LSB first

Pulse Distance Width Protokolle


Pulse Distance Width Coding

NUBERT

NUBERT	Wert
Frequenz	36 kHz?
Kodierung	Pulse Distance Width
Frame	1 Start-Bit + 10 Daten-Bits + 1 Stop-Bit
Daten	0 Adress-Bits + 10 Kommando-Bits ?
Start-Bit	1340µs Puls, 340µs Pause
0-Bit	500µs Puls, 1300µs Pause
1-Bit	1340µs Puls, 340µs Pause
Stop-Bit	500µs Puls
Wiederholung	einmalig nach 35ms
Tasten-Wiederholung	dritter, fünfter, siebter usw. identischer Frame
Bit-Order	MSB first?

SPEAKER

SPEAKER	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance Width
Frame	1 Start-Bit + 10 Daten-Bits + 1 Stop-Bit
Daten	0 Adress-Bits + 10 Kommando-Bits ?
Start-Bit	440µs Puls, 1250µs Pause
0-Bit	440µs Puls, 1250µs Pause
1-Bit	1250µs Puls, 440µs Pause
Stop-Bit	440µs Puls
Wiederholung	einmalig nach ca. 38ms
Tasten-Wiederholung	dritter, fünfter, siebter usw. identischer Frame
Bit-Order	MSB first?

ROOMBA

ROOMBA	Wert
Frequenz	38 kHz?
Kodierung	Pulse Distance Width
Frame	1 Start-Bit + 7 Daten-Bits + 0 Stop-Bit
Daten	0 Adress-Bits + 7 Kommando-Bits
Start-Bit	2790µs Puls, 930µs Pause
0-Bit	930µs Puls, 2790µs Pause
1-Bit	2790µs Puls, 930µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	dreimalig nach jeweils 18ms?
Tasten-Wiederholung	noch unbekannt
Bit-Order	MSB first

Biphase Protokolle


Biphase Coding

RC5 + RC5X

RC5 + RC5X	Wert
Frequenz	36 kHz
Kodierung	Biphase (Manchester)
Frame RC5	2 Start-Bits + 12 Daten-Bits + 0 Stop-Bits
Daten RC5	1 Toggle-Bit + 5 Adress-Bits + 6 Kommando-Bits
Frame RC5X	1 Start-Bit + 13 Daten-Bits + 0 Stop-Bit
Daten RC5X	1 invertiertes Kommando-Bit + 1 Toggle-Bit + 5 Adress-Bits + 6 Kommando-Bits
Start-Bit	889µs Pause, 889µs Puls
0-Bit	889µs Puls, 889µs Pause
1-Bit	889µs Pause, 889µs Puls
Stop-Bit	kein Stop-Bit
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	MSB first

RC6 + RC6A

RC6 + RC6A	Wert
Frequenz	36 kHz
Kodierung	Biphase (Manchester)
Frame RC6	1 Start-Bit + 1 Bit "1" + 3 Mode-Bits (000) + 1 Toggle-Bit + 16 Daten-Bits + 2666µs pause
Frame RC6A	1 Start-Bit + 1 Bit "1" + 3 Mode-Bits (110) + 1 Toggle-Bit + 31 Daten-Bits + 2666µs pause
Daten RC6	8 Adress-Bits + 8 Kommando Bits
Daten RC6A	"1" + 14 Hersteller-Bits + 8 System-Bits + 8 Kommando-Bits
Daten RC6A Pace (Sky)	"1" + 3 Mode-Bits ("110") + 1 Toggle-Bit(UNUSED "0") + 16 Bit + 1 Toggle(!) + 15 Kommando-Bits
Start-Bit	2666µs Puls, 889µs Pause
Toggle 0-Bit	889µs Pause, 889µs Puls
Toggle 1-Bit	889µs Puls, 889µs Pause
0-Bit	444µs Pause, 444µs Puls
1-Bit	444µs Puls, 444µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms
Bit-Order	MSB first

GRUNDIG + NOKIA

GRUNDIG + NOKIA	Wert

Frequenz	38 kHz (?)
Kodierung	Biphase (Manchester)
Frame-Paket	1 Start-Frame + 19,968ms Pause + N Info-Frames + 117,76ms Pause + 1 Stop-Frame
Start-Frame	1 Pre-Bit + 1 Start-Bit + 9 Daten-Bits (alle 1) + 0 Stop-Bits
Info-Frame	1 Pre-Bit + 1 Start-Bit + 9 Daten-Bits + 0 Stop-Bits
Stop-Frame	1 Pre-Bit + 1 Start-Bit + 9 Daten-Bits (alle 1) + 0 Stop-Bits
Daten Grundig	9 Kommando-Bits + 0 Adress-Bits
Daten Nokia	8 Kommando-Bits + 8 Adress-Bits
Pre-Bit	528µs Puls, 2639µs Pause
Start-Bit	528µs Puls, 528µs Pause
0-Bit	528µs Pause, 528µs Puls
1-Bit	528µs Puls, 528µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Info-Frames mit einem Pausenabstand von 117,76ms
Bit-Order	LSB first

IR60 (SDA2008)

IR60 (SDA2008)	Wert
Frequenz	30 kHz
Kodierung	Biphase (Manchester)
Start Frame	1 Start-Bit + 101111 + 0 Stop-Bits + 22ms Pause
Daten Frame	1 Start-Bit + 7 Daten-Bits + 0 Stop-Bits
Daten	0 Adress-Bits + 7 Kommando-Bits
Start-Bit	528µs Puls, 2639µs Pause
0-Bit	528µs Pause, 528µs Puls
1-Bit	528µs Puls, 528µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	keine
Tasten-Wiederholung	N-fache Wiederholung des Info-Frames mit einem Pausenabstand von 117,76ms
Bit-Order	LSB first

SIEMENS + RUWIDO

SIEMENS + RUWIDO	Wert
Frequenz	36 kHz? (Merlin-Tastatur mit Ruwido-Protokoll: 56 kHz)
Kodierung	Biphase (Manchester)
Frame Siemens	1 Start-Bit + 22 Daten-Bits + 0 Stop-Bits
Frame Ruwido	1 Start-Bit + 17 Daten-Bits + 0 Stop-Bits
Daten Siemens	11 Adress-Bits + 10 Kommando-Bits + 1 invertiertes Bit (letztes Bit davor nochmal invertiert)
Daten Ruwido	9 Adress-Bits + 7 Kommando-Bits + 1 invertiertes Bit (letztes Bit davor nochmal invertiert)
Start-Bit	275µs Puls, 275µs Pause
0-Bit	275µs Pause, 275µs Puls
1-Bit	275µs Puls, 275µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	1-malige Wiederholung mit gesetztem Repeat-Bit (?)
Tasten-Wiederholung	N-fache Wiederholung des Original-Frames innerhalb von 100ms (?)
Bit-Order	MSB first

A1TVBOX

A1TVBOX	Wert
Frequenz	38 kHz?
Kodierung	Biphase (Manchester) asymmetrisch

Frame	2 Start-Bits + 16 Daten-Bits + 0 Stop-Bits
Daten	8 Adress-Bits + 8 Kommando-Bits
Start-Bits	"10", also 250µs Puls, 150µs + 150µs Pause, 250µs Puls
0-Bit	150µs Pause, 250µs Puls
1-Bit	250µs Puls, 150µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	keine
Tasten-Wiederholung	unbekannt
Bit-Order	MSB first

ORTEK

ORTEK	Wert
Frequenz	38 kHz?
Kodierung	Biphase (Manchester) symmetrisch
Frame	2 Start-Bits + 18 Daten-Bits + 0 Stop-Bits
Daten	6 Adress-Bits + 2 Spezial-Bits + 6 Kommando-Bits + 4 Spezial-Bits
Start-Bit	2000µs Puls, 1000µs Pause
0-Bit	500µs Pause, 500µs Puls
1-Bit	500µs Puls, 500µs Pause
Stop-Bit	kein Stop-Bit
Wiederholung	2 zusätzliche Frames mit gesetzten Spezial-Bits
Tasten-Wiederholung	N-fache Wiederholung des 2. Frames
Bit-Order	MSB first

Pulse Position Protokolle

NETBOX

NETBOX	Wert
Frequenz	38 kHz?
Kodierung	Pulse Position
Frame	1 Start-Bit + 16 Daten-Bits, kein Stop-Bit
Daten	3 Adress-Bits + 13 Kommando-Bits
Start-Bit	2400µs Puls, 800µs Pause
Bitlänge	800µs
Wiederholung	keine
Tasten-Wiederholung	Abstand ca. 35ms?
Bit-Order	LSB first

Software-Historie IRMP

Änderungen IRMP in 2.6.x:

Version 2.6.7:

- 19.09.2014: Kleineren Bug behoben: Fehlendes Newline vor #else eingefügt

Version 2.6.6:

- 18.09.2014: Logging für ARM STM32F10X hinzugefügt

Version 2.6.5:

- 17.09.2014: PROGMEM-Zugriff für Array irmp_protocol_names[] korrigiert.

Version 2.6.4:

- 15.09.2014: Timing-Toleranzen für KASEIKYO-Protokoll vergrößert

Version 2.6.3:

- 15.09.2014: Wechsel von irmp_protocol_names auf PROGMEM, zusätzliche UART Routinen in main.c

Version 2.6.2:

- 21.07.2014: Portierung auf PIC 12F1840

Ältere Versionen:

- 09.07.2014: **Neues Protokoll**: SAMSUNG48
- 09.07.2014: Kleine Syntaxfehlerkorrektur
- 01.07.2014: Logging für ARM_STM32F4XX eingebaut
- 01.07.214: IRMP port für PIC XC8 compiler, Variadic Macros herausgenommen wg. dummen XC8-Compiler :-(
- 05.06.2014: **Neues Protokoll**: LGAIR
- 30.05.2014: **Neues Protokoll**: SPEAKER
- 30.05.2014: Timings für SAMSUNG-Protokolle optimiert
- 20.02.2014: Fehlerhaftes Decodieren des SIEMENS-Protokolls korrigiert
- 19.02.2014: **Neue Protokolle**: RCMM32, RCMM24 und RCMM12
- 17.09.2014: Timing für ROOMBA verbessert
- 09.04.2013: **Neues Protokoll**: ROOMBA
- 09.04.2013: Verbesserte Frame-Erkennung für ORTEK (Hama)
- 19.03.2013: **Neues Protokoll**: ORTEK (Hama)
- 19.03.2013: **Neues Protokoll**: TELEFUNKEN
- 12.03.2013: Geänderte Timing-Toleranzen für RECS80- und RECS80EXT-Protokoll
- 21.01.2013: Korrekturen Erkennung des Wiederholungsframes beim DENON-Protokoll
- 17.01.2013: Korrekturen Frame-Erkennung beim DENON-Protokoll
- 11.12.2012: **Neues Protokoll**: A1TVBOX
- 07.12.2012: Verbesserte Erkennung von DENON-Wiederholungsframes
- 19.11.2012: Portierung auf Stellaris LM4F120 Launchpad von TI (ARM Cortex M4)
- 06.11.2012: Korrektur DENON-Frame-Erkennung
- 26.10.2012: Einige Timer-Korrekturen, Anpassungen an Arduino
- 11.07.2012: **Neues Protokoll**: BOSE
- 18.06.2012: Unterstützung für ATtiny87/167 hinzugefügt
- 05.06.2012: Kleinere Korrekturen Portierung auf ARM STM32
- 05.06.2012: Include-Korrektur in irmpextlog.c (http://www.mikrocontroller.net/svnbrowser/irmp/irmpextlog.c?view=markup)
- 05.06.2012: Bugfix, wenn nur NEC und NEC42 aktiviert
- 23.05.2012: Portierung auf ARM STM32
- 23.05.2012: Bugfix Frame-Erkennung beim DENON-Protokoll
- 27.02.2012: Bug in IR60-Decoder behoben
- 27.02.2012: Bug in CRC-Berechnung von KASEIKYO-Frames behoben
- 27.02.2012: Portierung auf C18 Compiler für PIC-Mikroprozessoren
- 13.02.2012: Bugfix: oberstes Bit in Adresse falsch bei NEC-Protokoll, wenn auch NEC42-Protokoll eingeschaltet ist.
- 13.02.2012: Timing von SAMSUNG- und SAMSUNG32-Protokoll korrigiert
- 13.02.2012: KASEIKYO: Genre2-Bits werden nun im oberen Nibble von flags gespeichert.
- 20.09.2011: **Neues Protokoll**: KATHREIN
- 20.09.2011: **Neues Protokoll**: RUWIDO
- 20.09.2011: **Neues Protokoll**: THOMSON
- 20.09.2011: **Neues Protokoll**: IR60 (SDA2008)
- 20.09.2011: **Neues Protokoll**: LEGO
- 20.09.2011: **Neues Protokoll**: NEC16
- 20.09.2011: **Neues Protokoll**: NEC42
- 20.09.2011: **Neues Protokoll**: NETBOX
- 20.09.2011: Portierung auf ATtiny84 und ATtiny85
- 20.09.2011: Verbesserung von Tastenwiederholungen bei RC5
- 20.09.2011: Verbessertes Decodieren von Biphase-Protokollen
- 20.09.2011: Korrekturen am RECS80-Decoder
- 20.09.2011: Korrekturen beim Erkennen von zusätzlichen Bits im SIRCS-Protocol
- 18.01.2011: Korrekturen für SIEMENS-Protokoll
- 18.01.2011: **Neues Protokoll**: NIKON
- 18.01.2011: Speichern der zusätzlichen Bits (>12) im SIRCS-Protokoll in der Adresse
- 18.01.2011: Timing-Korrekturen für DENON-Protokoll

- 04.09.2010: Bugfix für F_INTERRUPTS >= 16000

- 02.09.2010: **Neues Protokoll**: RC6A

- 29.08.2010: **Neues Protokoll**: JVC
- 29.08.2010: KASEIKYO-Protokoll: Berücksichtigung der Genre-Bits. **ACHTUNG: dadurch neue Command-Codes!**
- 29.08.2010: KASEIKYO-Protokoll: Verbesserte Behandlung von Wiederholungs-Frames
- 29.08.2010: Verbesserte Unterstützung des APPLE-Protokolls. **ACHTUNG: dadurch neue Adress-Codes!**

- 01.07.2010: Bugfix: Einführen eines Timeouts für NEC-Repetition-Frames, um "Geisterkommandos" zu verhindern.

- 26.06.2010: Bugfix: Deaktivieren von RECS80, RECS80EXT & SIEMENS bei geringer Interrupt-Rate

- 25.06.2010: **Neues Protokoll**: RCCAR
- 25.06.2010: Tastenerkennung für FDC-Protokoll (IR-keyboard) erweitert

- 25.06.2010: Interrupt-Frequenz nun bis zu 20kHz möglich
- 09.06.2010: **Neues Protokoll**: FDC (IR-keyboard)
- 09.06.2010: Timing für DENON-Protokoll korrigiert
- 02.06.2010: **Neues Protokoll**: SIEMENS (Gigaset)
- 26.05.2010: **Neues Protokoll**: NOKIA
- 26.05.2010: Bugfix Auswertung von langen Tastendrücken bei GRUNDIG-Protokoll
- 17.05.2010: Bugfix SAMSUNG32-Protokoll: Kommando-Bit-Maske korrigiert
- 16.05.2010: **Neues Protokoll**: GRUNDIG
- 16.05.2010: Behandlung von automatischen Frame-Wiederholungen beim SIRCS-, SAMSUNG32- und NUBERT-Protokoll verbessert.
- 28.04.2010: Nur einige kosmetische Code-Optimierungen
- 16.04.2010: Sämtliche Timing-Toleranzen angepasst/optimiert
- 12.04.2010: **Neues Protokoll**: Bang & Olufsen
- 29.03.2010: Bugfix beim Erkennen von mehrfachen NEC-Repetition-Frames
- 29.03.2010: Konfiguration in irmpconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irmpconfig.h?view=markup) ausgelagert
- 29.03.2010: Einführung einer Programmversion in README.txt: Version 1.0
- 17.03.2010: **Neues Protokoll**: NUBERT
- 16.03.2010: Korrektur der RECS80-Startbit-Timings
- 16.03.2010: **Neues Protokoll**: RECS80 Extended
- 15.03.2010: Codeoptimierung
- 14.03.2010: Portierung auf PIC
- 11.03.2010: Anpassungen an verschiedene ATMega-Typen durchgeführt
- 07.03.2010: Bugfix: Zurücksetzen der Statemachine nach einem unvollständigen RC5-Frame
- 05.03.2010: **Neues Protokoll**: APPLE
- 05.03.2010: Die Daten irmp_data.addr + irmp_data.command werden nun in der jeweiligen Bit-Order des verwendeten Protokolls gespeichert
- 04.03.2010: **Neues Protokoll**: SAMSUNG32 (Mix aus SAMSUNG & NEC-Protokoll)
- 04.03.2010: Änderung der SIRCS- und KASEIKYO-Toleranzen
- 02.03.2010: SIRCS: Korrekte Erkennung und Unterdrückung von automatischen Frame-Wiederholungen
- 02.03.2010: SIRCS: Device-ID-Bits werden nun in irmp_data.command und nicht mehr in irmp_data.address gespeichert
- 02.03.2010: Vergrößerung des Scan Buffers (zwecks Protokollierung)
- 24.02.2010: Neue Variable flags in IRMP_DATA zur Erkennung von langen Tastendrücken
- 20.02.2010: Bugfix DENON-Protokoll: Wiederholungsframe grundsätzlich invertiert
- 19.02.2010: Erkennung von NEC-Protokoll-Varianten, z. B. APPLE-Fernbedienung
- 19.02.2010: Erkennung von RC6- und DENON-Protokoll
- 19.02.2010: Verbesserung des RC5-Decoders (Bugfixes)
- 13.02.2010: Bugfix: Puls/Pausen-Counter um 1 zu niedrig, nun bessere Erkennung bei Protokollen mit sehr kurzen Pulszeiten
- 13.02.2010: Erkennung der NEC-Wiederholungssequenz
- 12.02.2010: RC5-Protokoll-Decoder hinzugefügt
- 05.02.2010: Konflikt zwischen SAMSUNG- und MATSUSHITA-Protokoll beseitigt
- 07.01.2010: Erste Version

Software-Historie IRSND

Änderungen IRSND in 2.6.x:

Version 2.6.1:

- 10.07.2014: Einige GPIO Änderungen für STM32F10x (in IRSND).

Version 2.6.0:

- 10.07.2014: **Neues Protokoll**: SAMSUNG48

Ältere Versionen:

- 23.06.2014: **Neues Protokoll**: LGAIR
- 03.06.2014: **Neues Protokoll**: TELEFUNKEN
- 30.05.2014: **Neues Protokoll**: SPEAKER
- 30.05.2014: Timings für SAMSUNG-Protokolle optimiert
- 20.02.2014: **Neues Protokoll**: RUWIDO
- 09.04.2013: **Neues Protokoll**: ROOMBA
- 12.03.2013: 15kHz für RECS80- und RECS80EXT-Protokoll ist nun auch erlaubt
- 17.01.2013: Unterstützung für ATtiny44 hinzugefügt
- 12.12.2012: **Neues Protokoll**: A1TVBOX
- 07.12.2012: Korrektur Timing beim NIKON-Protokoll
- 26.10.2012: Einige Timer-Korrekturen, Anpassungen an Arduino
- 18.06.2012: Unterstützung für ATtiny87/167 hinzugefügt
- 05.06.2012: Korrekturen Portierung auf ARM STM32 - nun getestet
- 23.05.2012: Portierung auf ARM STM32 (ungetestet!)
- 23.05.2012: Bugfix Timing für 2. Frame beim Denon-Protokoll
- 27.02.2012: **Neues Protokoll**: IR60 (SDA2008)
- 27.02.2012: Bug beim Senden von Biphase-Frames (Manchester) behoben
- 27.02.2012: Portierung auf C18 Compiler für PIC-Mikroprozessoren
- 15.02.2012: Bugfix: Nur der 1. Frame wurde gesendet
- 13.02.2012: Timing von SAMSUNG- und SAMSUNG32-Protokoll korrigiert
- 13.02.2012: KASEIKYO: Genre2-Bits werden nun im oberen Nibble von flags gespeichert.
- 13.02.2012: Zusätzliche Pause nach dem Senden des letzten Frames
- 20.09.2011: **Neues Protokoll**: THOMSON
- 20.09.2011: **Neues Protokoll**: LEGO
- 20.09.2011: **Neues Protokoll**: NEC16
- 20.09.2011: **Neues Protokoll**: NEC42
- 20.09.2011: Portierung auf ATtiny84 und ATtiny85
- 20.09.2011: Korrektur von Pausenlängen
- 20.09.2011: Korrekturen von irsnd_stop()
- 20.09.2011: Korrektur des SIEMENS-Timings
- 20.09.2011: Umstellung auf 36kHz Modulationsfrequenz für DENON-Protokoll
- 20.09.2011: Korrektur Behandlung zusätzlicher Bits im SIRCS-Protokoll
- 18.01.2011: **Neues Protokoll**: RC6A
- 18.01.2011: **Neues Protokoll**: RC6
- 18.01.2011: **Neues Protokoll**: NIKON
- 18.01.2011: Beachten der zusätzlichen Bits (>12) im SIRCS-Protokoll
- 18.01.2011: Korrektur der Pausenlängen
- 18.01.2011: Timing-Korrekturen für DENON-Protokoll

- 02.09.2010: Neues Protokoll: JVC
- 02.09.2010: Anpassung des APPLE-Encoders an IRMP-Version 1.7.3.

- 29.08.2010: Neues Protokoll: KASEIKYO (Panasonic u.a.)

- 01.07.2010: Bugfix: Deaktivieren von RECS80, RECS80EXT & SIEMENS bei geringer Interrupt-Rate

- 25.06.2010: Neues Protokoll: RCCAR

- 09.06.2010: Neues Protokoll: FDC (IR-keyboard)
- 09.06.2010: Timing für DENON-Protokoll korrigiert

- 02.06.2010: Neues Protokoll: SIEMENS (Gigaset)
- 02.06.2010: Simulation von langen Tastendrücken

- 26.05.2010: Neues Protokoll: NOKIA

- 17.05.2010: Neues Protokoll: GRUNDIG
- 17.05.2010: Behandlung von Frame-Wiederholungen für SIRCS, SAMSUNG32 und NUBERT korrigiert

- 28.04.2010: Unterstützung des APPLE-Protokolls
- 28.04.2010: Konfiguration über irsndconfig.h (http://www.mikrocontroller.net/svnbrowser/irmp/irsndconfig.h?view=markup)

- 16.04.2010: Sämtliche Timing-Toleranzen angepasst/optimiert

- 14.04.2010: Neues Protokoll: Bang & Olufsen

- 17.03.2010: Neues Protokoll: NUBERT
- 17.03.2010: Korrektur der Pausen zwischen Frame-Wiederholungen

- 16.03.2010: Korrektur des Timer-Registers TCCR2
- 16.03.2010: Korrektur der RECS80-Startbit-Timings
- 16.03.2010: Neues Protokoll: RECS80 Extended

- 11.03.2010: Anpassungen an verschiedene ATMega-Typen durchgeführt

- 07.03.2010: Alpha-Version

Literatur

IR-Übersicht

- http://www.sbprojects.com/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 (IR Übersicht & RC5)

SIRCS-Protokoll

- http://www.sbprojects.com/knowledge/ir/sirc.php
- 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-Protokoll

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

LGAIR-Protokoll

Der LG Air Conditioner ist eine Klimaanlage, die durch eine "intelligente" Fernbedienung gesteuert wird. Dies sind die "entschlüsselten" Daten:

```
Befehl                  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 with 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-Protokoll (JVC)

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

SAMSUNG-Protokoll

(wurde aus diversen Protokollen (Daewoo u.ä.) zusammengereimt, daher kein direkter Link auf irgendwelche SAMSUNG-Dokumentation verfügbar)

Hier ein Link zum Daewoo-Protokoll, welches dasselbe Prinzip des Sync-Bits in der Mitte eines Frames nutzt, jedoch mit anderen Timing-Werten arbeitet:

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

MATSUHITA-Protokoll

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

KASEIKYO-Protokoll (auch "Japan-Protokoll")

- 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- und RECS80-Extended-Protokoll

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

RC5- und RC5x-Protokoll

- http://www.sbprojects.com/knowledge/ir/rc5.php
- 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-Protokoll

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

RC6 und RC6A-Protokoll

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

Bang & Olufsen

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

Grundig-Protokoll

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

Nokia-Protokoll

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

IR60 (SDA2008 bzw. 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-Protokoll

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

Diverse Protokolle

- 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 auf Youtube

Einige Videos zu IRMP habe ich auf Youtube gefunden:

- 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

Weitere Artikel zu IRMP

Whitepaper von Martin Gotschlich, Infineon Technologies AG (http://www.infineon.com/dgdl/RF2ir+WhitePaper+V1.0.pdf?folderId=db3a3043191a246301192dd3ee2c2ae4&fileId=db3a30432b57a660012b5c16272c2e81)

Hardware / IRMP-Projekte

Remote IRMP

Netzwerkfähiger Infrarot-Sender und Empfänger mit Android Handy als Fernbedienung:

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

IR-Tester

Eine Implementierung auf Basis IRMP und IRSND als Multiprotokoll Dekoder mit LCD von Klaus Leidinger:

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

IR-Tester mit AVR-NET-IO

Ähnliche Implementierung wie von Klaus Leidinger für Pollin AVR-NET-IO mit Pollin ADD-ON Board:

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

USB IR Remote Receiver

USB IR Remote Receiver von Hugo Portisch:

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

USB IR Empfänger/Sender/Einschalter mit 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-Einschalter auf Grundlage des USBasp

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

Servo-gesteuerter IR-Sender

Servo-gesteuerter IR-Sender mit Anlernfunktion von Stefan Pendsa:

- http://forum.mikrokopter.de/topic-21060.html
- SVN (http://svn.mikrokopter.de/listing.php?repname=Projects&path=%2FServo-Controlled+IR-Transmitter%2F&#Ad2417800d6aa14bf08c571a896e9def7)

Lernfähige IR-Fernbedienung

Lernfähige IR-Fernbedienung von Robert und Frank M.

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

AVR Moodlight

AVR Moodlight von Axel Schwenke

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

Kinosteuerung

Kinosteuerung von Owagner

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

Phasenanschnittsdimmer

Phasenanschnittsdimmer - steuerbar über IR-Fernbedienung:

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

IRDioder – Ikea Dioder Hack

Ikea Dioder Hack mit Atmel und Infrarotempfaenger:

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

Expedit Coffee Bar

Ikea Expedit Regal - umgebaut zur Kaffee-Bar:

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

Arduino als IR-Empfänger

Arduino als IR-Empfänger:

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

IR-Lautstärkesteuerung mit Stellaris Launchpad

IR-Lautstärkesteuerung mit Stellaris Launchpad (ARM Cortex-M4F):

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

RemotePi Board

Herunterfahren eines RaspPI mittels Fernbedienung:

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

Ethernut & IRMP

IRMP unter dem RTOS Ethernut:

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

LED strip Remote Control

LED-Beleuchtung per Fernbedienung steuern:

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

Ethersex & IRMP

IRMP + IRSND Modul in Ethersex, einer modularen Firmware für AVR MCUs

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

Mastermind Solver

Mastermind-Solver mit LED-Streifen und IR-Fernbedienung

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

IRMP + IRSND Library für STM32F4

IRMP für STM32F4

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

IRSND für STM32F4

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

Danksagung

Ganz herzlich bedanken möchte ich mich bei Vlad Tepesch, Klaus Leidinger und Peter K., die mich mit Scan-Dateien ihrer Infrarot-Fernbedienungen versorgt haben. Dank auch an Klaus für seine nächtelangen Tests von IRMP & IRSND.

Ebenso bedanken möchte ich mich bei Christian F. für seine Tipps zur PIC-Portierung. Vielen Dank auch an gera für die Portierung auf den PIC-C18 Compiler. Für die Portierung auf ARM STM32 bedanke ich mich herzlich bei kichi (Michael K.). Vielen Dank auch an Markus Schuster für die Portierung auf Stellaris LM4F120 Launchpad von TI (ARM Cortex M4).

Mein Dank geht auch an Dániel Körmendi, welcher mich nicht nur immer wieder fleißig mit Scans versorgt, sondern auch das LG-AIR-Protokoll in den IRSND eingebaut hat.

Diskussion

Meinungen, Verbesserungsvorschläge, harsche Kritik und ähnliches kann im Beitrag: Infrared Multi Protocol Decoder (http://www.mikrocontroller.net/topic/162119) geäussert werden.

Viel Spaß mit IRMP!

Kategorien: Infrarot | AVR-Projekte