IrpMaster: a program and API for the generation of IR signals from IRP notation

Table of contents

Note:

It may not be necessary to read this document. If your are looking for a user friendly GUI program for generating IR signals etc, please try the program IrScrutinizer (or its predecessor IrMaster), and get back here if (and only if) you want to know the detail on IR signal generation.

1 Revision history

Date	Description
2011-08-15	Initial version.
2012-04-24	Converted to the document format of Apache Forrest. The program documentation is now generated from that file. Many minor fixes and updates.
2012-06-03	Minor updates for upcoming release 0.2.0.
2012-08-19	Minor updates for upcoming release 0.2.1.
2012-11-18	Minor updates for upcoming release 0.2.2.
2014-01-27	Minor updates for upcoming release 1.0.0.

2 Revision notes

Release notes for the current version

3 Introduction

The "IRP notation" is a domain specific language for describing IR protocols, i.e. ways of mapping a number of parameters to infrared signals. It is a very powerful, slightly cryptic, way of describing IR protocols. In early 2010, Graham Dixon (mathdon in the JP1-Forum) wrote a specification. Up until this program was released, there has not been a usable implementation of the IRP-notation in the sense of a program that takes an IRP Protocol together with parameter values, and produces an IR signal. (The MakeHex program operates on a previous, more restricted version of the IRP notation. The MakeLearned program has severe restrictions, most importantly, its sources are not available.) The present work is a Java program/library that is hoped to fill that gap. It is written in Java 1.6, may or may not run with Java 1.5, but definitely not with earlier Java versions. It, optionally, calls the shared library DecodeIR on Windows, Linux, or Macintosh, but has no other "impurities" in the sense of Java. It can be used as a command line program, or it can be used through its API. For parsing the IRP-Notation, the tool ANTLR is used, generating the parser automatically from the grammar.

This project does not contain a graphical user interface (GUI). See Main principles for a background. Instead, the accompanying program IrScrutinizer (and its predecessor IrMaster) provides a GUI for the present program, among many other things.

For understanding this document, and the program, a basic understanding of IR protocol is assumed. However, the program can be successfully used just by understanding that an "IRP protocol" is a "program" in a particular "domain specific language" for turning a number of parameters into an IR signal, and the present program is a compiler/interpreter of that language. Some parts of this document requires more IRP knowledge, however.

3.1 Spelling, pronunciation

The mandatory section... :-; Preferred spelling is "IrpMaster", with "I" and "M" capitalized (just as the Java class). Pronounce it any way you like.

3.2 Synergies within other projects

I hope that this program/library should be useful to other projects involved in IR signals. It makes the vast knowledge of the JP1 project available to other programs. It can be used off-line, to manually or automatically produce e.g. configuration files containing rendered IR signal in some popular format, like the Pronto format. More exciting is to implement a real time "IR engine", that can generate and transmit IR signals in any of the known formats.

3.3 Copyright and License

The program, as well as this document, is copyright by myself. Of course, it is based upon the IRP documentation, but is to be considered original work. The "database file" IrpProtocols.ini is derived from DecodeIR.html, thus I do not claim copyright.

The program uses, or interfaces with (the different is slightly blurred), other projects. ExchangeIR was written by Graham Dixon and published under GPL3 license. Its Analyze-function has been translated to Java by myself, and is used in by the present program. DecodeIR was originally written by John S. Fine, with later contributions from others. It is free software with undetermined license. IrpMaster depends on the runtime functions of ANTLR3, which is free software with BSD type license.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

4 Main principles

4.1 Design principles

It is my opinion that it is better to get the functionality and the API right, before you do a graphical user interface (GUI). It is much easier and logical to put a GUI on top of a sane API, then to try to extract API functionality from a program that was never designed sanely but built around the GUI. (Look at WinZip for a good example of the latter. Or almost any Windows program, commercial or freeware...)

I have tried to follow the IRP document as closely as possible, in particular with respect to the grammar and the syntax. However, the Execution model of Chapter 14, turned out not to be usable.

Performance consideration were given minimal priorities. As it stands, rendering a single IR signal typically takes less than 1 ms, so this seems justified. Some debugging statements are covered by functionally superfluous if-statements in order not to have to evaluate arguments (`to String()` etc) not needed anyhow.

Everything that is a "time" is internally represented as a double precision number. Most output formats however, are in some integer format. I here use the principle of sound numerics, do all computations with "high precision" (i.e. double precision) as long as ever possible, then transform to the lower precision form (i.e. integer formats) only in the final step.

All "integer quantities" like expressions, are done in Java long format, 64 bits long, amounting to 63 bits plus sign. Already the `metanec`-example would not work with Java int's. The performance penalty over using `int` (32 bits) is believed to be neglectable.

Differently put, all parameters are limited to Java's long, and can thus be no larger than 2^63-1 = 9223372036854775807. A real-life protocol where this limit is exceeded is not known to me.

Versions prior to 0.2.1 also limited the length of bitfields, also the concatenation of bitfields, to 64. In version 0.2.1 this restriction has been removed, and arbitrary length (concatenation of) bitfields are allowed, as long as the parameters are less than 2^63-1. (Example: The concatenation of bitfields `A:50,B:50` produces a concatenated bitfield of length 100, which is accepted by IrpMaster 0.2.1, but rejected by prior versions.) (Thanx to 3FG for pointing this out to me.)

I do not have the slightest interest in internationalization of the project in its present form — it does not contain a user friendly interface anyhow.

4.2 Repetitions

Possibly the major difficulty in turning the IRP Specification into programming code was how to make sense of the repetition concept. Most treatises on IR signals (for example the Pronto format) considers an IR signal as an introduction sequence (corresponding to pressing a button on a remote control once), followed by a repeating signal, corresponding to holding down a repeating button. Any, but not both of these may be empty. In a few relatively rare cases, there is also an ending sequence, send after a repeating button has been released. Probably 99% of all IR signals fit into the intro/ repetition scheme, allowing ending sequence in addition should leave very few practically used IR signals left. In "abstract" IRP notation, these are of the form A,(B)+,C with A, B, and C being "bare irstreams".

In contrast, the IRP notation in this concept reminds they syntax and semantics of regular expressions: There may be any numbers, and they can even be hierarchical. There

certainly does not appear to be a consensus on how this very,... general ... notation should be practically thought of as a generator of IR signals. The following, "finite-automaton interpretation" may make sense: An IRP with several repetitions, say, A(B)+C(D)+E, can be thought of as a remote control reacting on single and double presses. Pressing the key and holding it down produces first A, then B's as long as the button is pressed. An action such as shortly releasing the key and immediately pressing it again then sends one C, and keeps sending D's as long as button is kept pressed. When released, E is sent. Similarly, hierarchical repetitions (repetitions containing other repetitions) may be interpreted with some secondary "key" being pressed and/or released while a "primary button" is being held down — possibly like a shift/meta/control modifier key on a keyboard or a sustain/ wah-wah-pedal on a musical instrument?

The present program does not implement hierarchical repetitions. However, an unlimited number of non-hierarchical repetitions are allowed, although not in the form if the class IrSignal — it is restricted to having three parts (intro, repeat, ending). Also, the repetition pattern (...)* is rejected, because it does not make sense as an IR signal.

The command line interpreter contains an "interactive mode", entered by the argument `--interactive`. This way the intrinsic finite state machine (see above) inherent in an IRP with repetitions can be interactively traversed, probably in the context of debugging.

5 Command line usage

5.1 Installing binaries

There is no separate binary distributions of IrpMaster. The user who do not want to compile the sources should therefore install the binary distribution of IrScrutinizer, which contains everything needed to run IrpMaster from the command line. Installing that package, either the Windows installer or the ZIP file, will install a wrapper, which is the preferred way to invoke IrpMaster.

5.2 Usage of the program from the command line

I will next describe how to invoke the program from the command line. Elementary knowledge of command line usage is assumed.

There is a lot of functionality crammed in the command line interface. The usage message of the program gives an extremely brief summary:

```
Usage: one of
        IrpMaster --help
        IrpMaster [--decodeir] [--analyze] [-c|--config <configfilename>] --version
        IrpMaster [OPTIONS] -n|--name <protocolname> [?]
        IrpMaster [OPTIONS] --dump <dumpfilename> [-n|--name <protocolname>]
        IrpMaster [OPTIONS] [--ccf] <CCF-SIGNAL>
        IrpMaster [OPTIONS] [-n|--name] <protocolname> [PARAMETERASSIGNMENT]
        IrpMaster [OPTIONS] [-i|--irp] <IRP-Protocol> [PARAMETERASSIGNMENT]
```

```
where OPTIONS=--stringtree <filename>,--dot <dotfilename>,--xmlprotocol
 <xmlprotocolfilename>,
-c|--config <configfile>,-d|--debug <debugcode>|?,-s|--seed <seed>,-q|--quiet,
-P|--pass <intro|repeat|ending|all>,--interactive,--decodeir,--analyze,--lirc
 <lircfilename>,
-o|--outfile <outputfilename>, -x|--xml, -I|--ict, -r|--raw, -p|--pronto, -u|--uei,
--disregard-repeat-mins, -#|--repetitions <number_repetitions>.

Any filename can be given as `-', meaning stdin or stdout.
PARAMETERASSIGNMENT is one or more expressions like `name=value' (without spaces!).
One value without name defaults to `F`, two values defaults to `D` and `F`,
three values defaults to `D`, `S`, and `F`, four values to `D`, `S`, `F', and `T`, in
 the order given.

All integer values are nonnegative and can be given in base 10, 16 (prefix `0x'),
8 (leading 0), or 2 (prefix `0b' or `%'). They must be less or equal to 2^63-1 =
 9223372036854775807.

All parameter assignment, both with explicit name and without, can be given as
 intervals,
like `0..255' or '0:255', causing the program to generate all signals within the
 interval.
Also * can be used for parameter intervals, in which case min and max are taken from
the parameterspecs in the (extended) IRP notation.
```

Note that if using the wrapper previously described, it has already added the option `--config` *`standard_confile`* to the command line.

We will next explain this very brief description somewhat more verbosely:

- The first version simply produces the help message, as per above.
- The second version will will print the versions of the program, and optionally, the version of the configuration file and the DecodeIR dynamic library. The `--version` argument should normally be given last, since it is executed immediately when the command line is parsed.
- The third version prints the IRP string of the protocol with the given name to the terminal.
- In the forth version, a CCF string (or, alternatively, a string in UEI learned format) is read in, and, depending on the to the other options invoked, translated to another format, or sent to DecodeIR and/or AnalyzeIR.
- The fifth version dumps either the whole IRP data base, or just the protocol given as argument, to the file name used as the argument to the `--dump` option (use `-` for standard output).
- The sixth version uses the name of an IRP protocol (using the `-n` or `--name` option), to be found in the data base/configuration file specified by the `-c` or `--config` option, that protocol is used to render an IR signal or sequence using the supplied parameters (more on that later).
- Finally, the last version allows the user to enter an explicit IRP string using the `-i` or `--irp`-option, to be used to render the signal according to the parameters given.

In the simplest and most general form, parameter assignments are made on the command line in one argument of the type *`name=value`*. On both sides of the "="-signs, there

should not be any spaces. (More precisely, it is required that all assignments are made within a single "argument" to the program, which is determined by the command line interpreter. Thus writing the arguments within single or double quotes, extra spaces can be parsed.) After named parameters are given (possibly none), up to four "standard" parameters can be given. These are, in order D, S, F, and T (which per convention in the JP1 community stands for "Device", "Subdevice", "Function" (also called OBC or command number), and "Toggle"). If using -1 as the value, that parameter is considered as not being assigned. One value without name defaults to `F', two values defaults to `D' and `F', three values defaults to `D', `S', and `F', and four to `D', `S', `F', and `T', in the order given. For example,

```
E=12 34 -1 56 1
```

assigns the value 12 to E, the value 34 to D, the value of 56 to F, and 1 to T, while S is not assigned anything at all. Parameters can be given not only in decimal notation, but also as hexadecimal (using prefix 0x) binary (using prefix 0b or %), or octal (using prefix 0).

If the command line cannot be parsed the usage message will be printed. If you are unsure of exactly what is wrong, consider issuing "`-d 1`" (the debug option with argument 1) as the first argument on the command line, which may produce more verbose error messages.

Using the `-r` or `--raw` option, the output is given in "raw form" (in JP1-Forum jargon, this is a sequence of positive numbers (indicating "flashes", or on-times in micro seconds) and negative numbers (indicating "gaps" or off-times, where the absolute value indicates the duration in micro seconds. Carrier frequency is specified separately). Alternatively, or additionally, using the `-p` or `--pronto` option, output is produced in the so-called Pronto format, see e.g. this document. This format is popular in several IR using Internet communities, like Promixis (known for their (commercial) products Girder and NetRemote), as well as EventGhost. Optionally, these can be wrapped into an XML skeleton, offering an ideal platform for translating to every other IR format this planet has encountered. If desired, the output of the program is directed to a particular named file using the `-o` *`filename`* or `--output` *`filename`* option. (There is also a possibility (using the `--ict` or `-I` option) to generate output files in IRScope's ict-format, but I am not sure this was as wise design decision: it may be a better idea to generate additional formats by post-processing the XML file.)

5.2.1 Preventing intro sequence in repeat sequence

Motivated by this thread in the JP1 forum, I have been thinking over the "correct" way to render signals of this type `... (...)+`. This is a real issue, to determine the correct behavior when e.g. a program is sent the instruction "send the signal one time", and not an academic question like "keypress shorter than 6ms" or de-bouncing circuitry.

The Pronto notation is normally described as "intro part exactly once, repetition part if and as long as the button is held down". I.e., zero or more times. Therfore, IMHO,

the IRP `I (R)+` should properly be rendered as having intro sequence `I R`, which is what IrpMaster normally does. However, in a sense, this can be considered as ugly, awkward, and redundant. If I recall properly, there is a flag in the LIRC configuration called something like "send_repeat_least_once", which should be exactly what we need.

The option called `--disregard-repeat-mins` will make IrpMaster render the intro sequence without repetition part, also in the `... (...)+` case.

5.3 Iterating over input parameter ranges

Either for generating configuration files for other programs, or for testing, there is a very advanced feature for looping over input parameter sets. For all of the parameters to a protocol, instead of a single value, a set can be given. The program then computes all IR signals/sequences belonging to the Cartesian product of the input parameter sets. There are five types of parameter sets:

1. Of course, there is the singleton set, just consisting of one value
2. There is also a possibility to give some arbitrary values, separated by commas. Actually, the commas even separate sets, in the sense of the current paragraph.
3. An interval, optionally with a stride different from 1, can be given, either as `min..max++increment` or `min:max++increment`, or alternatively, simply as *, which will get the min and max values from the parameter's parameter specs.
4. Also, a set can be given as `a:b<<c`, which has the following semantics: starting with `a`, this is shifted to the left by `c` bits, until `b` has been exceeded (reminding of the left-shift operator << found in languages such as C).
5. Finally, `a:b#c` generates c pseudo random numbers between a and b (inclusive). The "pseudo random" numbers are completely deterministically determined from the seed, optionally given with the `--seed` option. As of version 0.2.2 `a` and `b>` are optional. If left out, the values are taken as from the protocol parameters min and max respectively, just as with the `*` form.

See the test file `test.sh` (include in the distributions) for some examples. Of course, using the command line, some of the involved characters, most notably the *, has a meaning to the command line interpreter and may need "escaping" by a backslash character, or double or single quotes.

There is also an option, denoted -# or `--repetitions` taking an integer argument, that will compute that many "copies" of the IR signal or sequence. This may be of interest for signals that are non-constant (toggles being the simplest example) or for profiling the program.

5.4 Debugging possibilities

There are a number of different debug parameters available. Use `-d` or `--debug` with "?" as argument for a listing:

```
$ java -jar IrpMaster.jar --debug ?
```

```
Debug options: Main=1, Configfile=2, IrpParser=4, ASTParser=8, NameEngine=16,
 BitFields=32, Parameters=64, Expressions=128,
IrSignals=256, IrStreamItems=512, BitSpec=1024, DecodeIR=2048, IrStreams=4096,
BitStream=8192, Evaluate=16384
```

For every debug option, there is an integer of the form 2^n associated with it. Just add the desired numbers together and use as argument for the `-d` or `--debug` command. There are also commands for debugging the parsed version of the IRP: Notably the `--stringtree` *`filename`* option (produces a LISP-like parsed representation of the so-called AST (abstract syntax tree)). `--dot`*`filename`* produces a dot-file, that can be translated by the open-source program `dot` contained in the Graphviz project, producing a nice picture (e.g. in any common bitmap format) of the current IRP protocol, and `--xmlprotocol` *`filename`* producing an XML representation. It may be possible in the future to use any of these representations to e.g., write a C code generator for a particular protocol.

Some of the classes contain their own main methods (for those not familiar with the Java jargon: these can be called as programs on their own) allowing for both debugging and pedagogical exploration, together possibly with other possibilities. In particular, this goes for the Expression class, One day I am going to document this...

```
java -classpath IrpMaster.jar org.harctoolbox.IrpMaster.Expression -d  'a + b
 *c**#d' {a=12,b=34,c=56,d=4}
(+ a (* b (** c (BITCOUNT d))))
1916
```

5.5 Third-party Java archives (jars)

For the DecodeIR-integration, IrpMaster requires a small support package, DecodeIR.jar, which is distributed together with IrpMaster. It consists of the compiled `DecodeIRCaller.java` from DecodeIR (full name `com.hifiremote.decodeir.DecodeIRCaller.class`), and `com.hifiremote.LibraryLoader.class` from RemoteMaster, which is also free software. To get rid of some (in this context) annoying messages, it was necessary to create a (very) lightly modified version, which can be found on the download page. IrpMaster also requires the runtime libraries of the parser generator ANTLR, which is also free software but licensed under a BSD-License. I distribute the whole (binary) package `antlr-3.4-complete.jar`. (No usable "runtime version" is known to me.)

6 Extensions to, and deviation from, IRP semantic and syntax

6.1 Parameter Specifications

In the first, now obsolete, version of the IRP notation the parameters of a protocol had to be declared with the allowed max- and min-value. I have reinvented this, using the name parameter_spec. For example, the well known NEC1 protocol, the Parameter Spec reads: `[D:0..255,S:0..255=255-D,F:0..255]`. (D, S, and F have the semantics of

device, sub-device, and function or command number.) This defines the three variables D, S, and F, having the allowed domain the integers between 0 and 255. D and F must be given, however, S has a default value that is used if the user does not supply a value. The software requires that the values without default values are actually given, and within the stated limits. If, and only if, the parameter specs is incomplete, there may occur run-time errors concerning not assigned values. It is the duty of the IRP author to ensure that all variables that are referenced within the main body of the IRP are defined either within the parameter specs, defined with "definitions" (Chapter 10 of the specification), or assigned in assignments before usage, otherwise a run-time error will occur (technically an `UnassignedException` will be thrown).

The preferred ordering of the parameters is: D, S (if present), F, T (if present), then the rest in alphabetical order,

The formal syntax is as follows, where the meaning of the '@' will be explained in the following section:

```
parameter_specs:
 '[' parameter_spec (',' parameter_spec )* ']' | '['  ']'

parameter_spec:
 name ':' number '.' '.' h=number ('=' i=bare_expression)?
 | name '@' ':' number '.' '.' number '=' bare_expression
```

6.2 The GeneralSpec

For the implementation, I allow the four parts (three in the original specification) to be given in any order, if at all, but I do not disallow multiple occurrences — it is quite hard to implement cleanly and simply not worth it. (For example, ANTLR does not implement exclusions. The only language/grammar I know with that property is SGML, which is probably one of the reasons why it was considered so difficult (in comparison to XML) to write a complete parser.)

6.2.1 Persistency of variables

Graham, in the specification and in following forum contributions, appears to consider all variables in a IRP description as intrinsically persistent: They do not need explicit initialization, if they are not, they are initialized to an undefined, random value. This may be a reasonable model for a particular physical remote control, however, from a scientific standpoint it is less attractive. I have a way of denoting a variable, typically a toggle of some sort, as persistent by appending an "@" to its name in the parameter specs. An initial value (with syntax as default value) is here mandatory. It is set to its initial value by the constructor of the Protocol class. Calling the renderIrSignal(...) function or such of the Protocol instance typically updates the value (as given in an assignment, a 0-1 toggle goes like T=1-T). As opposed to variables that has not been declared as persistent, it (normally) retains its value between the invocations of renderIrSignal(...). A toggle is typically declared as `[T@:0..1=0]` in the parameter specs.

6.2.2 Comments and line breaks

Comments in the C syntax (starting with `/*` and ended by `*/`) are allowed and ignored. Line breaks can be embedded within an IRP string by "escaping" the line break by a backslash

6.2.3 Data types

The IRP documentation clearly states that the carrier frequency is a real number, while everything else is integers. Unfortunately, users of the IRP notation, for example in the DecodeIR.html document, has freely used decimal, non-integer numbers. I have implemented the following convention: Everything that has a unit (second or Hz), durations and frequency, are real numbers (in the code double precision numbers).

6.2.4 Extents

The specification writes ``*An extent has a scope which consists of a consecutive range of items that immediately precede the extent in the order of transmission in the signal. ... The precise scope of an extent has to be defined in the context in which it is used.*", and, to my best knowledge, nothing more. I consider it as specification hole. I have, starting with IrpMaster 0.2.2, implemented the following: Every extend encountered resets the duration count.

6.2.5 Multiple definitions allowed

It turned out that the preprocessing/inheritance concept necessitated allowing several definition objects. These are simply evaluated in the order they are encountered, possibly overwriting previous content.

6.2.6 Names

Previous programs (makehex, makelearned) have only allowed one-letter names. However, in DecodeIR.html there are some multi-letter names. The IRP documentation allows multi-letter names, using only capital letters. I have, admittedly somewhat arbitrarily, extended it to the C-name syntax: Letters (both upper and lower cases) and digits allowed, starting with letter. Underscore "_" counts as letter. Case is significant.

Also there are a few predefined, read-only variables, mainly for debugging, although a practical use is not excluded. To distinguish from the normal, and not to cause name collision, they start by a dollar sign. Presently, these are: `$count` (numbers the call to a `render*-()`-function, after the constructor has been called), `$pass`(Requested pass in a `--pass`-argument, (or from API call), not to be confused with the following), `$state` (current state (intro=0, repeat=1, ending=2,...) of parsing of an IRP), `$final_state` (undefined until the final state has been reached, then the number of the final state). For example, the OrtekMCE example `{...}<...>([P=0][P=1]`

`[P=2],4,-1,D:5,P:2,F:6,C:4,-48m)+[...]` could be written with `$state` as `(4,-1,D:5,$state:2,F:6,C:4,-48m)+` (disregarding last frame).

6.2.7 GeneralSpecs, duty cycle

Without any very good reason, I allow a duty cycle in percent to be given within the GeneralSpec, for example as `{37k,123,msb,33%}`. It is currently not used for anything, but preserved through the processing and can be retrieved using API-functions. If some, possibly future, hardware needs it, it is there.

6.2.8 Namespaces

There is a difference in between the IRP documentation and the implementation of the Makehex program, in that the former has one name space for both *assignments* and *definitions*, while the latter has two different name spaces. IrpMaster has one name space, as in the documentation. (This is implemented with the `NameEngine` class.)

6.2.9 Shift operators (not currently implemented)

It has sometimes been suggested (see this thread) to introduce the shift operators "<<" and ">>" with syntax and semantics as in C. This far, I have not done so, but I estimate that it would be a very simple addition. (The reader might like to have a look at my example, which possibly would have been more naturally expressed with left shifts than with multiplication with powers of two.)

6.2.10 Logical operators (also not implemented)

In particular in the light of current discussion on the F12 protocol, in my opinion more useful would be the logical operators `&&`, `||`, and `?:`, having their short circuiting semantics, like in languages such as C, Perl,..., but unless, e.g. Pascal. Recall, the expression `A && B` is evaluated as follows: First `A` is checked for being 0 or not. If 0, then 0 is returned, without even evaluating `B`. If however, `A` is nonzero, `B` is evaluated, possibly to a "funny" type and is returned. The F12 protocol (cf. the latest version 2.43 of DecodeIR.html) could then probably be written like `<...>(introsequence, (H && repetitionsequence*))` or `<...>(H ? longsequence+ : shortsequence)`.

6.2.11 BitCount Function

Generally, I think you should be very reluctant to add "nice features" to something like IRP. However, in the applications in DecodeIR.html, the phrase "number of ones", often modulo 2 ("parity"), occurs frequently in the more complicated protocols. This is awkward and error prone to implement using expressions, for example: `F:1 + F:1:1 + F:1:2 + F:1:3 + F:1:4 + F:1:5 + F:1:6 + F:1:7`. Instead, I have

introduced the BitCount function, denoted by "#". Thus, odd parity of `F` will be `#F%1`, even parity `1-#F%2`. It is implemented by translating to the Java Long.bitCount-function.

6.3 Preprocessing and inheritance

Reading through the protocols in DecodeIR.html, the reader is struck by the observation that there are a few general abstract "families", and many concrete protocol are "special cases". For example all the variants of the NEC* protocols, the Kaseikyo-protocols, or the rc6-families. Would it not be elegant, theoretically as well as practically, to be able to express this, for example as a kind of inheritance, or sub-classing?

For a problem like this, it is easily suggested to invoke a general purpose macro preprocessor, like the C preprocessor or m4. I have successfully resisted that temptation, and am instead offering the following solution: If the IRP notation does not start with "{" (as they all have to do to confirm with the specification), the string up until the first "{" is taken as an "ancestor protocol", that has hopefully been defined at some other place in the configuration file. Its name is replaced by its IRP string, with a possible parameter spec removed — parameter specs are not sensible to inherit. The process is then repeated up until, currently, 5 times.

The preprocessing takes place in the class IrpMaster, in its role as data base manager for IRP protocols.

6.3.1 Example

This shows excepts from a virtual configuration file. Let us define the "abstract" protocol `metanec` by

```
[protocol]
name=metanec
irp={38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)[A:0..4294967295]
```

having an unspecified 32 bit payload, to be subdivided by its "inherited protocols". Now we can define, for example, the NEC1 protocol as

```
[protocol]
name=NEC1
irp=metanec{A = D | 2**8*S | 2**16*F | 2**24*(~F:8)}[D:0..255,S:0..255=255-D,F:0..255]
```

As can be seen, this definition does nothing else than to stuff the unstructured payload with D, S, and F, and to supply a corresponding parameter spec. The IrpMaster class replaces "`metanec`" by `{38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)`" (note that the parameter spec was stripped), resulting in an IRP string corresponding to the familiar NEC1 protocol. Also, the "Apple protocol" can now be formulated as

```
[protocol]
name=Apple
```

```
irp=metanec{A=D | 2**8*S | 2**16*C:1 | 2**17*F | 2**24*PairID} \
{C=1-(#F+#PairID)%2,S=135} \
[D:0..255=238,F:0..127,PairID:0..255]
```

The design is not cast in iron, and I am open to suggestions for improvements. For example, it seems reasonable that protocols that only differ in carrier frequency should be possible to express in a concise manner.

6.4 The Configuration file/IRP protocol database

There is presently not a "official" IRP database. MakeHex comes with a number of protocol files with the `.irp`-extension, but that is another, obsolete and much less powerful format. MakeLeaned also comes with a number of "irp-files", in the new format, but incomplete. The DecodeIR.html-file presently comes closest: it has a number (upper two-digit) of IRPs, however, often not even syntactically confirming to the specification, and often with the description of the protocol at least partially in prose ("C is the number of ..."), parseable only by humans, not by programs.

Possibly as an intermediate solution, I invented the `IrpProtocols.ini` file. This file has a format similar to ini-files under Windows. For every protocol, it contains name and an IRP-string, possibly also a documentation string. The latter can, in principle, contain HTML elements, i.e. it can be an HTML fragment.

6.5 Syntax and semantics of the IrpProtocols.ini file

Every protocol is described in a section starting with the key `[protocol]`. Then there are a few keywords describing different properties:

- `name` The name of the protocol. This is folded to lowercase for searches and comparisons.
- `irp` The IRP string representation. This may continue over several lines if the line feeds are escaped by a backslash ("\"), i.e. having the backspace as last character on the line.

Other keywords are allowed, but ignored. Then, optionally, there may be a section `[documentation]`, that, in principle, could contain e.g. an HTML-fragment. The documentation section continues until the next `[protocol]` is encountered.

6.6 Requirements for an IRP data base

I have created the present `IrpProtocols.ini` by hand editing the DecodeIR.html-file. I would welcome if the community can settle for one endorsed format for such a data base. It can be one file, or many files: One file per protocol is easier for the developer, in particular if several developers are using a version management system (with or without file locking), but less convenient for the user.

It would be highly desirable in the future to be able just to maintain one file (or set of files). Some possibilities for this are:

1. Have one master file, for example in XML format, that after preprocessing *generates* both DecodeIR.html, and a protocol description file. There is also the possibility of having a program like IrpMaster parsing the master file directly.
2. Extend `protocol.ini`("belonging to RemoteMaster") with the IRP information. Leaves the problem of duplicated "documentation" between DecodeIR.html and protocols.ini.
3. Formalizing the IRP-Strings within DecodeIR.html, e.g. by using div or span elements with class-attributes, (and formatting with, for example, better CSS style sheets) so that the IRP information can be unambiguously read out.

6.7 Integration with DecodeIR

Optionally (when installed and selected with the `--decodeir` option) the computed IR signal is sent to DecodeIR, to check DecodeIR's opinion on the nature of the signal. This gives a magnificent possibility for automated tests, not only of the present program, but also of DecodeIR. Note in particular that there are very advance possibilities for testing not only a single signal, but for testing whole ranges of signals, a list of signals, "random" inputs, equidistant inputs, or inputs achieved by shifting, see the section on parameter iterating.

The shared library is sought first in architecture dependent sub-directories, like in RemoteMaster, `.\windows` on Windows, `./Linux-amd64` and `./Linux-i386` on 64- and 32-bit Linux respectively, etc, then in system libraries, for example given on the command line to the Java VM, using the `-Djava.library.path=` option.

There is some fairly hairy programming in `DecodeIR.java` for identifying some different cases.

The enclosed script `test.sh` runs under a Unix/Linux shell such as bash or sh. It should also run within Cygwin on Windows. It does not run with the standard Windows command line interpreter. Note that it might need some adjustment of file paths etc.

Possibly because I did not find any more logical way to dispose it, the current distribution contains a class (with main()-method) named EvaluateLog that can be used to evaluate the output of the above script. Use like

```
java -classpath IrpMaster.jar IrpMaster/EvaluateLog protocols.log
```

7 The API

The Java programmer can access the functionality through a number of API functions.

The class `IrpMaster` is the data base manager. The class is immutable, constructed from a file name (or an InputStream), and can deliver assorted pieces of information from the data base. Most interesting is the `newProtocol()`-function that generates a Protocol-object from parsing the IRP-string associated with the requested protocol name. It contains a very elaborate `main()`-function for command line use -- strictly speaking

this is "the program" that is described herein. Actually, that `main()`-function does not necessarily belong to the `IrpMaster` class, but could be located somewhere else.

Instances of the Protocol class are constructed (essentially) from a String, containing the IRP representation to be parsed. Once constructed (and IRP-String parsed), the Protocol instances can render IrSignals and IrSequences for many different parameter values. This is done with the `render(...)` and `renderIrSignal(...)` functions, producing `IrSequences` and `IrSignals` respectively:

An `IrSequence` is a sequence of pulse pairs. It does not know weather it is supposed to repeat or not. In contrast, an `IrSignal` has one introductory `IrSequence`, one repetition `IrSequence` (either, but not both, of these can be empty), and an (in most cases empty) ending `IrSequence`.

The API is documented in standard Javadoc style, which can be installed from the source package, just like any other Java package. For the convenience of the reader, the Javadoc API documentation is also available here.

7.1 Example of API usage

The task is to write a command line program, taking, in order, the configuration file name, a protocol name, a device number and a function/command/obc number, and send the corresponding IR signal to a GlobalCaché GC-100-06 networked IR-transmitter, having IP address 192.168.1.70, using its IR Port 1. For this, we use the GlobalCaché functionality of the Harctoolbox, which is also GPL-software written by myself. This task is solved with essentially just a few lines of code.

8 References

1. IrScrutinizer. A program, also by myself, than, among other things, provides a user friendly GUI for IrpMaster.
2. IrMaster. A program, also by myself, than, among other things, provides a user friendly GUI for IrpMaster.
3. Specification of IRP Notation, Graham Dixon. Also in PDF version for download. A very thorough specification.
4. Discussion thread on the IRP documentation
5. DecodeIR.html. (The link points to a slightly nicer formatted wiki page, though). Contained within the current distribution of DecodeIR.
6. Makehex. Source, binary. A functional predecessor of the present program. Operates on a predecessor of the current version of the IRP. Written in C++, also available as DLL (within the first link). Java translation by myself.
7. MakeLeaned. Windows, binary only, source unavailable. GUI only, no API. Not maintained since 2005. Almost certainly incomplete with respect to current IRP specification. Discussion thread in JP1-Forum.