Specification of IRP Notation

Second draft, 4 February 2010

The notation currently used for describing IR protocols, generally just referred to as "IRP notation", was developed by John Fine in a series of postings on a restricted Yahoo forum on 13 March 2003 and a few following days. Partial descriptions have appeared since in various messages on the JP1 website, but I have been unable to find a comprehensive description. I feel it is time for one to appear.

This document attempts to be a full specification in sufficient detail that an automated process ro parse and execute a protocol in IRP notation could be written without the fear that some possible construction has been overlooked. The sections on syntax give an informal description followed by a formal specification of the syntax and a description of the semantics of that syntax. A later section of the document gives an execution model written in the context of a conceptual remote control. This structure leads to some inevitable duplication in places, but I think the benefits outweigh the disadvantages.

1. General principles of IRP notation

1.1 Introduction

The function of an IR protocol is to transmit a set of named numerical values from one device to another, normally from a remote control to the piece of equipment that it controls, by flashing an infrared light on and off. The IR Protocol (IRP) notation is a means of describing how this is to be done. The names of the values are commonly "Device code", "Subdevice code" and "Function code", but some or even all of these may be absent and others may be present. There may also be values for other purposes, such as

- a calculated "Checksum" to provide a confirmation that the signal has been received correctly, or
- a "Toggle" that the sender changes from one signal to the next to enable the receiver to distinguish two otherwise identical signals from one longer signal.

The infrared signal consists of a series of flashes of light with gaps in between. In a flash the light is usually not on continuously, but is being pulsed on and off rapidly at a fixed frequency, known as the *carrier frequency*. In the gaps, the light is fully off. As an example, a flash lasting 1200 microseconds could consist of 48 on/off cycles at a frequency of 40kHz and could be followed by a gap of 600 microseconds before the next flash. The information is carried in the durations of the flashes and gaps. The frequency itself carries no information. Frequencies vary between protocols, but are fixed for any individual protocol.

In the language of IR protocols a flash and its following gap is generally known as a "burst pair". The IRP notation describes the encoding process by which the values to be transmitted are converted into a stream of burst pairs with varying durations.

1.2 Basic objects

The basic objects of the IRP notation are *names* and *numbers*. *Names* are sequences of capital letters and digits but must not start with a digit. Numbers are sequences of digits but must not start with zero unless they consist solely of the digit zero. The notation is composed of names and numbers together with other

characters from the ASCII character set. White space, e.g. space characters, tabs, line feeds and form feeds, are ignored but may be used anywhere for convenience and ease of readability.

Names and *numbers* both ultimately represent *values*. Names are used for two purposes. *External names* are given values by means outside the IRP notation. These values are the data that the protocol is to convey. *Internal names* correspond to checksums or any other derived values. *Internal names* are given values by means of *definitions* that show how they are to be calculated.

For conciseness it is common to use single capital letters for names, with the correspondence

D = Device code
S = Subdevice code
F = Function code (otherwise called OBC or Original Button Code)
C = Checksum
T = Toggle.

Other single letters are also used as required, but so are abbreviations such as OEM1.

The values represented by *names* and *numbers* are integers. Values are usually positive integers or zero, but negative integer values can also occur, such as in the value of a name that represents displacement of a mouse. The value of a number is that number regarded as an integer expressed in decimal notation. Negative integers cannot be represented directly by *numbers*. A negative integer has to be enclosed in parentheses, e.g. (-12). This is a simple example of an *expression*. It is not a basic object as it is composed of the number 12 together with the non-alphanumeric symbols "(", "-" and ")".

1.3 The execution process

Execution is the process of generating the IR signal from the representation of a protocol in IRP notation. There is a specification of the execution process in section 14 in terms of the action of a conceptual remote control. The resulting signal is described in terms of the durations of the flashes and gaps that compose it, together with the frequency and duty cycle of the carrier wave that is sent during a flash.

The carrier frequency was described in 1.1. The *duty cycle* is the percentage of the cycle time that the IR light is on. A carrier frequency of 40kHz means that there are 40000 cycles per second, so each lasts 25µs. If each cycle consists of the IR light being on for 9µs and off for 16µs then the duty cycle is 9/25, or 36%. The frequency varies substantially between one protocol and another and is given in the IRP notation. The duty cycle is a matter for the implementation and is not given in the IRP notation. Duty cycles are typically around 33%. A larger value means that the batteries in a remote control will last less long but the range of the signal will be greater.

1.4 Execution concepts

There are two intermediate concepts used in the specification of the execution process, a *binary form* and a *bit sequence*. These do not have specific IRP notations as they do not occur as such in the IRP representation of protocols. A *binary form* is essentially the type of representation of an integer value that is used internally in computers, namely the value expressed in binary notation with a fixed bit length. The four-bit binary form for the value 5, for example, is 0101, and for -5 is 1011 (as these two add to 0000 when overflow is ignored).

A binary form has a numerical value. This is its value treated as the binary representation of an integer on the assumption that any leading bits not included in the binary form are 0's. It may seem pedantic to be

this specific, but it is necessary since the value of a finite length binary form that represents a negative integer is not that integer, it is a positive integer which depends on that length. The value of the 4-bit binary form for -5 given above is 11, the value of the 6-bit binary form for -5 is 59.

Binary forms of infinite length can occur conceptually during execution. These are the only binary forms whose value can be negative. They occur implicitly in computers in the Arithmetic Shift Right operation on the binary representation of signed integers. In this operation the vacant space created on the left by the shift is filled not with zero but with a copy of the most significant bit of the unshifed value, as if that bit value continued indefinitely to the left and one bit of it has just been shifted into visibility.

The other intermediate concept in execution is a *bit sequence*. This is a sequence of bits given in the order in which they are to be processed for transmission. It is therefore a sequence in time, but when written the sequence order is from left to right. A bit is a binary digit, as in the binary notation for integers, so a written bit sequence looks very much like an integer in binary form. They are not the same, however, and the distinction is very important in IRP execution. The bits in a bit sequence have no interpretation in themselves. They are constructed from the values to be sent, and values are reconstructed from them when received, but it is the structure of the protocol and not the rules of binary notation that determines the relationship between values and bit sequences. In particular there is no demarcation between consecutive bit sequences constructed from distinct values.

During the execution process, values are translated into binary forms, binary forms into bit sequences and bit sequences into the sequences of flash and gap durations that describe the signal, so in outline the steps of the execution process are

values → binary forms → bit sequences → duration sequences

1.5 Translation objects

The rule for each of these translation steps in the execution process is specified by a particular type of object in IRP notation. The translation of values to binary forms is described by a *bitfield*, that of binary forms to bit sequences by a *sequencing rule* and that of bit sequences to duration sequences by a *bitspec*.

These object differ in their scope. The scope of a *bitfield* is an individual value. The scope of a *sequencing rule* is the entire protocol. The scope of a *bitspec* is in between these two extremes. It covers everything up to the point at which another bitspec is given that changes the rule.

Not all durations in the signal come from values in this way. There are also flashes and gaps in the signal that are for control purposes, such as to distinguish one protocol from another. These do not depend on the data. They are described by specific notations for a *flash duration* and a *gap duration*.

There remains one other type of translation object. This is a gap of variable duration that is used to make the total extent in time of the signal, or some major part of the signal, up to a constant length that is independent of the data being sent. It is of variable duration since the lengths of the individual flashes and gaps do depend on the data. The notation for such a variable gap is an *extent*. The typical use of extents is as the lead-out gap at the end of the signal. Many buttons on a remote control repeat the signal throughout the time that the button is held down. Using an extent as a lead-out ensures that the frequency of repeats does not depend on the data being sent. In the translation phase the data is known, so that an extent can be translated into a specific *gap duration*

1.6 Overall structure

The complete description of a protocol in IRP notation is composed of three or four sections:

- A *GeneralSpec* that specifies the *carrier frequency*, the *sequencing rule* and a time unit that can be used in *durations* and *extents*;
- A *Bitspec* that provides the default rule for the translation of *bit sequences* to *duration sequences*.
- An *IRStream* that gives the sequence of data to be translated and transmitted.
- If required, *Definitions* that give the expressions required to calculate the values of *internal names*.

These are all described in detail in the following sections. The IRstream section can contain further bitspecs and IRstreams but there is only one GeneralSpec and at most one Definitions section. The Bitspec section provides only a default rule as bitspecs within the IRstream section may change the rule for certain parts of the data.

1.7 Formal syntax

Each descriptive part of the specification will be accompanied by a formal syntax given in EBNF (Extended Backus-Naur Form) notation and by a precise description of the semantics. An execution model that incorporates these semantics is given in section 14.

There are various extensions of BNF notation in use. This document uses the version standardized by ISO in ISO/IEC 14977:1996, which is freely available on the ISO website.

The characters used in the IRP notation are all characters of the International Reference Version of ISO/IEC 646:1991, commonly referred to as ASCII. Certain characters have more than one usage in the IRP notation, distinguished from one another by context. In particular, the LESS-THAN SIGN and GREATER-THAN SIGN also serve as left and right diamond brackets respectively and the HYPHEN-MINUS is used both as a minus sign and as a symbol with a meaning specific to IRP notation.

The formal syntax of the overall structure is:

```
protocol = generalspec, bitspec, irstream, [definitions];
```

In EBNF notation, square brackets denote an optional item. There are no semantics at this level.

2. The GeneralSpec section

2.1 Description

The GeneralSpec section specifies the carrier frequency and certain data that apply throughout the translation process. It is a list of up to three items, separated by commas and enclosed in curly brackets. Three types of item can occur in the list. Each type is optional and there can be at most one item of each type. They are:

- A frequency item, consisting of a number followed by the letter "k". It gives the *carrier frequency* in kilohertz. The default is a frequency of "0k", which signifies a baseband signal in which the light stays on continuously during a flash. Although optional, a frequency item is

normally given since the default value is uncommon and so is best specified explicitly when it applies.

- A unit item, consisting of a number followed optionally by either of the letters "u" or "p". It determines a unit of time that can be used in *durations* and *extents*.
  - No suffix, or a suffix "u" indicates that the unit is that number of microseconds.
  - A suffix "p" indicates that the unit is the time taken for that number of pulses of the carrier frequency, so the unit in microseconds is the given value multiplied by (1000/f), where f is the number preceding the "k" in the frequency item.
  - If there is no unit item, it defaults to "1u", i.e. one microsecond.
- An order item, which gives the *sequencing rule* that applies globally for the conversion of binary forms to bit sequences. If present it takes one of the values "lsb" or "msb". If it is "lsb" then the time ordering of the bits in the bit sequence is that the sequence starts with the least significant bit ("lsb"). If it is "msb" then the ordering is to start with the most significant bit. The default is "lsb". In contrast to the situation with the frequency item, the default is the most common rule and it is not usually specified explicitly.

The number in a frequency item can include a decimal point. This is the only situation in IRP notation in which numbers can be other than integers.

These items can be given in any order. Examples of possible GeneralSpecs are:

{ }, {38.4k, 564}, {msb, 889u}, {10p, msb, 40k}

In the first of these, all items take their default value, so the signal is baseband, the sequencing rule is to convert binary forms to bit sequences in lsb order and the time unit is one microsecond. In the last of these, the carrier frequency is 40 kilohertz, binary forms are converted in msb order and the time unit is 10*(1000/40) = 250 microseconds.

2.2 Formal syntax

The complication in giving a formal syntax for the GeneralSpec section comes from the order of the item types being arbitrary. This is dealt with by allowing each item to be of any type but then imposing a restriction that no two items can be of the same type, as follows:

```
generalspec = "{", generalspec list, "}";
generalspec list = | (generalspec item, 2*[",", generalspec item])
   - generalspec list with repeats;
generalspec item = frequency item | unit item | order item;
frequency item = number with decimals, "k";
unit item = number, ["u" | "p"];
order item = "lsb" | "msb";

generalspec list with repeats =
   {generalspec item, ","}, frequency item, ",", {generalspec item, ","},
   frequency item, {",", generalspec item}
 | {generalspec item, ","}, unit item, ",", {generalspec item, ","},
   unit item, {",", generalspec item}
 | {generalspec item, ","}, order item, ",", {generalspec item, ","},
   order item, {",", generalspec item};
```

Vertical bars separate alternatives. The vertical bar at the start of `generalspec list` is preceded by an empty alternative, signifying that a `generalspec list` can be empty. The other alternative is one

`generalspec item` followed by two more optional ones each preceded by a comma separator, provided that the list so constructed is not also a `generalspec list with repeats`. This consists of all lists of items that include two of the same type, specified separately for each possible repeated type. Curly brackets in EBNF notation denote an arbitrary number of repetitions, including zero.

The undefined identifier `number` is the basic object described in 1.2. Its formal definition is given in 13.2 The identifier `number with decimals` means the same except that a decimal point is allowed.

2.3 Semantics

The real number in a frequency item is the carrier frequency in kilohertz. A zero frequency means that the signal is baseband, i.e. during a flash the IR light is on continuously. This is the default when this item is absent.

A unit item specifies a length of time that can be used as a time unit in durations and extents. If it has no suffix or the suffix "u" then `number` is the length of time in microseconds. If it has suffix "p" then the length of time in microseconds is `number` multiplied by (1000/*f*) where *f* is the real number preceding the "k" in the frequency item. The result of the multiplication is rounded to the nearest integer. If the unit item is absent then it specifies a time unit of one microsecond.

The order item specifies the global *sequencing rule*. If it is absent or has the value "lsb" then binary forms are converted to bit sequences by starting with the least significant bit. If it has value "msb" then they are converted starting with the most significant bit.

3 Durations

3.1 Description

Durations specify a length of time in the sending of the IR signal. They describe the modulation of the signal carrier and have a meaning only as part of a sequence. There are two types of duration, a *flash duration* and a *gap duration*. A flash duration is a length of time for which the signal carrier is "on". A gap duration is a length of time for which it is "off". These are the only two states of the carrier during a signal; it is either fully on or fully off.

The notation for a *flash duration* is simply the length of time, given in terms of a time unit determined by a suffix to the numerical value as follows:

- "m" indicates that the duration is in milliseconds;
- "u" indicates that the duration is in microseconds;
- "p" indicates that the duration is in pulses of the carrier frequency, as described in 2.1 above;
- no suffix indicates that the duration is in the time unit determined by the unit item of the GeneralSpec section.

The notation for a *gap duration* is the same but prefixed by "-". If there are two consecutive flash durations or gap durations in a sequence then the effect is the same as a single one for the total of the two times. There is nothing in the transmitted signal to indicate where one ends and the next starts.

Suppose that the GeneralSpec is {40k, 200}. Then if A = 150, the very artificial sequence

15p,-1m,3,Au,-20m

means:

- a flash of 15*(1000/40) = 375 microseconds, followed by
- a gap of 1 millisecond = 1000 microseconds, followed by
- a flash of 3*200 + 150 = 750 microseconds, followed by
- a gap of 20 milliseconds = 20000 microseconds.

This illustrates that units can be mixed, that names can be used and can have suffixes and that flashes and gaps do not have to alternate.

3.2 Formal syntax

```
duration = flash duration | gap duration;
flash duration = name or number, ["m"|"u"|"p"];
gap duration = "-", name or number, ["m"|"u"|"p"];
name or number = name | number;
```

3.3 Semantics

A flash duration is a length of time in an IR signal for which the carrier is in the ON state. A gap duration is a length of time for which it is in the OFF state.

The suffix determines how the value of the name or number is converted into a length of time in microseconds, as follows:

- For suffix "u" the number is already the time in microseconds.
- For suffix "m", multiply by 1000.
- For suffix "p", multiply by (1000/*f*) where *f* is the real number preceding the "k" in the frequency item of the GeneralSpec.
- If there is no suffix, multiply by the length in microseconds of the time unit specified by the unit item of the GeneralSpec.

4. Extents

4.1 Description

An *extent* is a gap duration whose length in time cannot be given explicitly in the protocol. 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 role of an extent is to ensure that the time taken for this range of items to be transmitted, *plus* the following gap (corresponding to the extent), has a constant value even though the time taken by the individual items may vary according to the data being transmitted. It is this constant value that is given in the extent. The precise scope of an extent has to be defined in the context in which it is used.

The notation for an extent is as that for a gap duration, except that the prefix is "^" instead of "-". When all the items in its scope have been translated into durations, the extent can be translated into a gap

duration by subtracting the total length in time of the durations in its scope from the time given in the extent. An artificial example is

1,-4,D,^25

as an extent with its preceding scope. This translates to 1,-4,10,-10 if D = 10 but to 1,-4,5,-15 if D = 5, the final gap making the duration as a whole up to 25 units regardless of the value of D.

4.2 Formal syntax

```
extent = "^", name or number, ["m"|"u"|"p"];
```

4.3 Semantics

The name or number together with the suffix, if any, determine a length of time as in the semantics of a duration described in 3.3.

An extent has a scope that has to be defined in any context in which an extent can be used. During execution, all the items in the scope are converted into durations. The extent then translates into a gap duration by subtracting the total length in time of these durations from the length of time determined by the extent.

5. Bitfields

5.1 Structure and interpretation

A bitfield is a translation object that converts a specific value to a *binary form*. Note, however, that the *binary form* may be constructed from only a portion of the bits in the binary expression of the value and that even if it corresponds to all these bits then the value of the binary form may not be the same as the value from which it was constructed.

A simple example of a bitfield is D:6. The *binary form* consists of the least significant six bits of D, with the binary representation of D considered as padded on the left with potentially an infinite number of zeroes. So if D = 244, its value in binary notation is ...00011110100 and the least significant six bits give the binary form 110100. If D = 5, its value in binary notation is ...00000101 and the least significant six bits give the binary form 000101.

A bitfield such as D:6:2 is similar, but leaves out the two least significant bits and selects the next six. For D = 244 and D = 5 this gives respectively the binary forms 111101 and 000001. The bitfield D:6 is equivalent to D:6:0.

The value concerned can be negative. In this case no padding is required. The complete binary representation of a negative integer (in the usual 1's complement form) extends on the left to infinity with all tbe bits beyond a certain point being 1. So -5 is ...11111011 where the dots denote an infinite sequence of ones (since when this is added to ...000000101 the digits of the sum are all zero). If D = -5, the binary form given by D:6 is 111011. To write this using an explicit value, we must use brackets, thus (-5):6 to show that the minus sign is part of the value and not a prefix of the IRP notation. This is an example of the use of an *expression* in a bitfield. Expressions are described in more detail in 9.1 below.

There are three further refinements to the bitfield notation. The first element of the bitfield can have a prefix "~". This complements each bit in the binary form, i.e. changes a 1 to a 0 and vice versa. The second element can be absent or, if present, can have a prefix "-". If it is absent then the bit length of the binary form is conceptually infinite, as described above. In this case the third element must be present to prevent the bitfield ending with a colon and so giving rise to ambiguous syntax. A third element can always be added without changing the interpretation, so although D: is not valid, D::0 is equivalent and is valid. Prefixing the second element with "-" reverses the order of the bits in the binary form, so the most significant bit becomes the least significant bit, etc. So with D = 244, ~D:6:2 gives the binary form 000010, D:-6:2 gives 101111 and ~D:-6:2 gives 010000.

It was explained in 1.4 that binary forms have values, although bit sequences do not. With D = 244 the values of these three binary forms are ~D:6:2 = 2, D:-6:2 = 47 and ~D:-6:2 = 16.

There are still more subtleties to be aware of. Each element can be any type of *primary item*, a primary item being a name, a number or an expression, provided that the second and third elements evaluate to non-negative integers. Expressions can be recognised as they are enclosed in parentheses but they can themselves contain bitfields. When a bitfield occurs in an expression, its value is the value of the binary form to which it translates, not the binary form itself. So F:D is a valid bitfield whose binary form consists of the D least significant bits of the value of F. It is not valid to take D = -4 and expect this to be the same as F:-4, since the prefix "-" in F:-4 is part of the IRP syntax, not of the value of the second element. But F:-D with D = 4 is valid, and does mean F:-4. If A = 7 then F:A:2 means F:7:2 but F:(A:2) means F:3, since A = 111 in binary notation so A:2 is the binary form 11 and the expression (A:2) evaluates to 3.

5.2 Formal syntax

```
bitfield = ["~"], primary item, ":", (["-"], primary item, [":", primary
   item] | ":", primary item);
primary item = name | number | expression;
(* The different types of primary item can be distinguished by their first
   character as follows:  digit => unsigned integer, capital letter => name,
   "(" => expression. *)
```

5.3 Semantics

Each type of primary item evaluates to an integer value. A bitfield evaluates to a *binary form*, an integer value expressed in binary notation with a fixed finite or infinite bit length in which negative integers are expressed in 1's complement notation. The value of the bitfield *a*:*b*:*c*, where the integer *a* and non-negative integers *b* and *c* are the values of the primary items, is determined as follows:

- express *a* in its binary representation, conceptually padded on the left with an infinite number of 0's or 1's as appropriate, negative values being expressed in the usual 1's complement form;
- delete the *c* least significant bits;
- if *b* is present take the next *b* least significant bits and discard the rest, otherwise *b* is absent and nothing is discarded, the binary form is the conceptually infinite form that remains.

The value of *a*:*b* is the same as *a*:*b*:0. In the general case where either or both of the permitted prefixes are present then apply the following further steps:

- if the first item in the bitfield has the prefix "~" then complement each bit in the sequence, i.e. replace 1's by 0's and 0's by 1's;

- if the second item in the bitfield has the prefix "-" then reverse the order of the sequence.

Note that the "-" prefix is not permitted unless *b* is present. If either *b* or *c* is negative then the bitfield is invalid. In the execution process, a bitfield is a translation object that translates the value *a* to a binary form.

6. IRstreams

6.1 Description

As seen in 1.6, the detail of an IR protocol is carried in an IRstream but that is not the only role of IRstreams, as will be seen in section 7. An IRstream consists of a list of items that represents an order for transmission, together with an optional marker that describes whether and how those items are to be repeated in the transmission. Each item ultimately translates to a duration or a sequence of durations. IRstreams can be nested but the outer one, the IRstream section described in 1.6, translates to the sequence of durations that describes the entire signal.

Each item in the list is one of the following:

- Duration
- Extent
- Assignment
- Variation
- Bitfield
- IRstream
- Bitspec followed by IRstream

The notation for an IRstream is that the items are separated by commas, the list is enclosed in parentheses and optionally followed by a *repeat marker*. Repeat markers are the subject of section 8 below.

Items that are durations need no translation. Extents are translated into durations in accordance with section 4, the *scope* of an extent being all items from the start of the IRstream to the extent itself. Assignments do not themselves translate to durations, instead they change the values bound to names in the global environment. Variations are alternative sections of the IRstream in which the active alternative may vary as the IRstream is repeated in accordance with its repeat marker. Durations and extents have been described in previous sections, assignments and variations are described in sections 11 and 12 The scope rule for an extent is handled automatically by the execution model of section 14.

Bitspecs also have a specific scope. Every bitspec is always followed by an IRstream, but not necessarily conversely. As the list of item types shows, an IRstream can contain other IRstreams that do not have preceding bitspecs. The *scope* of a *bitspec* is the whole of the IRstream that follows it but excluding the scopes of any bitspecs that this IRstream contains.

A bitfield is processed as follows. Its primary items (names, numbers and expressions) are evaluated in the context of the global environment. The bitfield is then evaluated as a binary form as described in 5.3. The sequencing rule determined by the GeneralSpec in accordance with 2.3 is then applied to translate the binary form into a bit sequence. Adjacent bit sequences are concatenated and then processed further in accordance with the currently active bitspec, i.e. the one in whose scope they lie. This is described in section 7 below.

During execution the processing of an IRstream is performed in left to right order, i.e. in the order of its items for transmission. This is significant as assignments may change the values bound to names in the global environment, so that subsequent items are processed in the context of the changed environment. When this sequential processing reaches an item that is itself an IRstream, processing of the current IRstream is suspended and that of the new one commenced. When the processing of this new IRstream is completed, that of the previous one is resumed. In the execution model of section 14, this is handled through a last in, first out (LIFO) IRstream stack. When the sequential processing reaches an item that is a bitspec, this becomes the currently active bitspec. When processing of the IRstream that follows this bitspec is completed, the bitspec that was formerly active is restored as the current one. In the execution model of section 14, this is handled through a LIFO bitstream stack.

As a simple example of an IRstream, consider the following one that occurs in the Dish Network:

(F:-6,U:5,D:5,1,-15).

In this protocol the sequencing rule is lsb. Suppose F=13, U=3, D = 25. In binary notation these values are F = 1101, U = 11, D = 11001. Pad these to the lengths specified in the bitfields to give F = 001101, U = 00011, D = 11001. Take account of the "-" prefix in F:-6 to give the binary forms

F:-6 = 101100, U:5 = 00011, D:5 = 11001.

As the sequencing rule is lsb, these translate separately to the bit sequences

0 0 1 1 0 1, 1 1 0 0 0, 1 0 0 1 1

where the bits have been spaced out to distinguish them from binary forms. As these are adjacent in the IRstream they are concatenated into the single bit sequence

0 0 1 1 0 1 1 1 0 0 0 1 0 0 1 1

for further processing by the active bitspec.

There is now no demarcation in the bit sequence to show where the bits from the different external values start and end. This comes into its own if the bitspec translates bits in groups rather than individually, a possibility that will be seen in section 7. In the Dish Network protocol they actually are translated individually, but suppose for the sake of this example that they are translated in groups of four, as actually happens in the XMP protocol. This grouping would be

0 0 1 1, 0 1 1 1, 0 0 0 1, 0 0 1 1

which is very different from that resulting from the binary forms. The bits comprising the second of these groups are actually the two least significant bits of F in msb order, followed by the two least significant bits of U in lsb order, but this group would be processed by the bitspec as a single entity. In a similar way both U and D contribute to the third group. As a result there would be no part of the IR signal that could be said to be the encoding of any of F, U or D. These separate values can be recovered from the signal only in the context of the protocol specification that was used to translate and encode them.

The concatenation of bit sequences and the subsequent re-grouping for processing by a bitspec is handled in the execution model by means of a first in, first out (FIFO) bit buffer. Each bitspec has its own such buffer for the bits it is to process.

6.2 Formal syntax

```
irstream = "(", bare irstream, ")", [repeat marker];
bare irstream = | irstream item, [",", bare irstream -];
(* This allows a bare irstream to be empty but the bare irstream in the
   second alternative cannot be empty.  This is the significance of the
   minus sign, which is the EBNF exception symbol.  What is excluded by
   the exception is what follows it, which in this case is empty. *)
irstream item = duration | extent | assignment | variation | bitfield |
   irstream | bitspec, irstream;
```

6.3 Semantics

The items of a bare irstream are processed sequentially from left to right, each in accordance with its own semantics. In the case of a bitfield there is a further processing stage. Processing of a bitfield yields a binary form. This is then translated to a bit sequence in accordance with the global sequencing rule.

The permitted item types include extent and bitspec, each of which requires its scope to be specified in any context in which it can be used. Within an irstream the scope of a bitspec is every item in the following irstream except for items of the form "bitspec, irstream", where the scope includes the bitspec but not the irstream. The scope of an extent is every item that precedes it in that irstream.

7. Bitspecs

7.1 Description

A *bitspec* specifies a rule for translating a bit sequence into a bare IRstream, i.e. an IRstream without its enclosing parentheses and any following repeat marker. In effect, this bare IRstream simply replaces the bit sequence to create a new IRstream. The details of this process are given below.

The action of a bitspec is not restricted to a specific bit sequence. Instead it is a general rule that has a well-defined *scope* specified in 6.1 and which applies to any bit sequence in its scope.

The rule usually translates each bit separately but it may translate bit pairs or bit quadruples, or indeed bits taken in groups of any size but only groups of 1, 2 or 4 bits have yet been seen in real protocols. The rule is defined simply by giving the bare IRstream corresponding to every possible combination of 0's and 1's in a group of the appropriate size.

For a bitspec that translates bits in groups of n, the number of possible combinations is 2 to the power n. The corresponding sequences of bare IRstreams are listed in the order of the bit sequences given by the bitfields

0:n, 1:n, 2:n, 3:n, ... (n-1):n.

These bitfields are translated to bit sequences as if they were themselves IRstream items, so this is a two-step process. First they are translated to n-bit binary forms and then the global sequencing rule is applied to further translate them to bit sequences. If the sequencing rule is "msb" and n = 3 then this order is

0 0 0, 0 0 1, 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0, 1 1 1

but if the sequencing rule is "lsb" then the order is

0 0 0, 1 0 0, 0 1 0, 1 1 0, 0 0 1, 1 0 1, 0 1 1, 1 1 1.

The notation for a bitspec is a list of the bare IRstreams separated by vertical bars "|" with the whole enclosed in diamond brackets (i.e. less-than and greater-than signs). The number of bare IRstreams does not have to be a power of two, but if it is not then it is treated as if this number were the next greater non-zero power of two and with the bare IRstreams for the later bit sequences being empty. It is not a syntax error as a bare irstream is permitted to be empty. The qualification "non-zero" permits, say, <1,-2> to be a valid bitspec for which $n = 1$, the bit 0 is mapped to the bare IRstream 1,-2 and the bit 1 is mapped to the empty bare IRstream.

Some examples of bitspecs from real protocols are:

<1,-1,1,-8|1,-10>, <-1,1|1,-1>, <1,-3.3m|1,-4.7m>, <-4,2|-3,1,-1,1|-2,1,-2,1|-1,1,-3,1>.

These illustrate the form that is by far the most common, in that the bare IRstreams are simply sequences of durations. Bitfields also occur in bitspecs within IRstreams for some real protocols. The IRP notation, however, permits any of the types of item listed in 6.1 as permitted in IRstreams and the execution model of section 14 can handle any such construction.

The action of a bitspec in the execution process is as follows. Recall that adjacent bit sequences are concatenated, as described in 6.1, so this description applies to the resulting concatenated sequence. The bits of the bit sequence to be processed are taken in groups of the appropriate size starting from the beginning of the sequence. Each group is then translated into the corresponding bare IRstream. This will result in a series of consecutive bare IRstreams that are themselves concatenated into a single, larger, one. The overall effect is therefore to translate a bit sequence into a single bare IRstream. The execution process then processes this bare IRstream before returning to process the subsequent items of the original IRstream.

In the general form of a bitspec the bare IRstream that it constructs may itself contain bitfields (which occurs in a number of real protocols) or even IRstreams possibly proceded by other bitspecs (which have not yet been seen in real protocols). In either case the whole process for the translation of bitfields to bare irstreams may be called recursively. This raises the question of which bitspec is used for such recursive processing. This is answered by the scope rule for bitspecs. As described in 6.1, when a bitspec is encountered within an IRstream, the scope of the new bitspec is the IRstream that follows it. The bare IRstreams within that new bitspec fall within the scope of the "old" bitspec that applies to the containing IRstream, and therefore it is this "old" bitspec that applies in recursive processing. The execution model of section 14 handles this through the bitspec stack that was mentioned in 6.1.

7.2 Execution example 1: Zenith protocol

As an introduction to the execution model of section 14, this subsection examines a simple but non-trivial example, the Zenith protocol, to see how the semantics described so far can be modelled in terms of stacks and buffers. The protocol is

{40k,520}<1,-1,1,-8|1,-10>(S:1,<1:2|2:2>(F:D),-90m)+

where the plus sign is a repeat marker that will be explained in 8.1.

Suppose that D = 4, S = 1 and F = 43. The first item to be processed is the outer bitspec. This gets pushed on the bitspec stack and becomes the currently active one. The next one is the outer IRstream, which first gets pushed on the irstream stack and then its contents is processed.

This processing begins with the bitfield S:1, which now is 1:1 and which translates to the trivial bit sequence 1. The currently active bitspec translates this to the bare IRstream

1,-10

which in the execution model is then transmitted as the start of the signal.

The execution process then meets the inner bitspec, which is pushed on the bitspec stack and becomes the currently active one. The following IRstream is pushed on the IRstream stack and flagged as having a preceding bitspec.

The contents of this inner IRstream is now processed. It has one item, the bitfield F:D. The binary representation of F is 101011, so as a binary form F:D = 1011. The sequencing rule is lsb, so the bit sequence is 1 1 0 1. This is pushed into the buffer for the active (inner) bitspec, but as there are no more items in the IRstream then the next step is to process this buffer with the bitspec to which is belongs.

The first stage in processing the buffer is to make the previous bitspec (the outer one) be the active one by decrementing the stack pointer. The bits are pulled and processed (FIFO) one at a time, the order being 1 1 0 1. The first bit gets mapped to the bitfield 2:2, translated to the binary form 10 and then, since the sequencing rule is lsb, to the bit sequence 0 1. This is pushed into the buffer of the currently active bitspec, which is now the outer one. Note that two buffers are now in current use.

The remaining three bits of the buffer being processed are pulled, mapped and processed similarly; in general this continues until either the buffer is empty or processing of the bare IRstream resulting from the mapping meets something other than a bitfield. This is the way the concatenation of bit sequences is handled. When either of these happen then the buffer is processed. In the present case this continues until the buffer being processed is empty and the buffer of the current (outer) bitspec contains

0 1 0 1 1 0 0 1.

This buffer is now processed by its bitspec. The bitspec stack pointer is again decremented, which means that there is now no current bitspec. So if a bitfield was encountered it would be an error, but that does not happen here as each of these bits is mapped to a sequence of durations by the outer bitspec according to the mapping.

0 → 1,-1,1,-8 1 → 1,-10.

In the execution model these are transmitted as they are encountered. When this has been done and this buffer is empty, the previous active bitspec, the outer one, is again made current by incrementing the bitspec stack pointer. Its buffer is now empty so the bitspec stack pointer is incremented once again to make the inner bitspec be active.

Processing of the inner IRstream now resumes, with the item after F:D. But the end of this IRstream has been reached, so its processing has been completed, together with that of its preceding bitspec since its scope was just this IRstream. As the IRstream was flagged has having a preceding bitspec, that bitspec is pulled from the stack, once again making the outer bitspec be active. The completed (inner) IRstream is also pulled from its stack and processing resumes of the outer IRstream.

There is only one item left, the lead-out gap -90m. Execution waits for this time and then continues with the next execution of the (outer) IRstream in accordance with the repeat marker.

7.3 Execution example 2: Proton protocol

Extents do not occur in the Zenith protocol, but a real example in which they do occur is the Proton protocol. This protocol is

{38k,500}<1,-1|1,-3>(16,-8,D:8,1,-8,F:8,1,^63m)+

where again the plus sign is a repeat marker.

To handle extents in the execution model, when each IRstream is pushed on the stack its start time is recorded. If an extent is encountered while processing that IRstream, execution adds its time period to the start time and waits until that is the current time. In this way both the scope of the extent and its conversion to a gap duration get incorporated simply into the model.

Suppose the values of the device code D and function code (OBC) F are D = 34, F = 19. There are no nested IRstreams or bitspecs so processing is straightforward. The bitspec is pushed on the bitspec stack, the IRstream on the IRstream stack and the start time is recorded, say T. The initial durations 16, -8 are converted to a flash and gap and transmitted as the start of the signal. Since in binary notation D = 100010 and the sequencing rule is lsb, the bitfield D:8 translates to the bit sequence

0 1 0 0 0 1 0 0

This is pushed into the buffer of the bitspec but is immediately processed since the next item in the bare IRstream is a duration, not a bitfield. The bitspec shows that the bits are translated to duration sequences one at a time, with the correspondence

0 → 1,-1 1 → 1,-3.

The corresponding flashes and gaps are sent as the next part of the signal, followed by those for the two durations 1, -8 that are next in the bare IRstream. The bitfield F:8 translates to the bit sequence

1 1 0 0 1 0 0 0

which is mapped to durations just as for D:8 and sent next since it is followed by another duration, value 1, which follows these as a flash.

At this point the time taken by the flashes and gaps already transmitted totals 76 units. The unit is that specified by the GeneralSpec, namely 500µs, so 76 units is 38 milliseconds. The real-time clock that read T at the start of the processing of this IRstream therefore now reads T+38 milliseconds. The extent value is 63 milliseconds so the execution model waits until the current time is T+63 milliseconds before it performs the next action (which would be a repeat execution of the IRstream). This is a gap of 25 milliseconds, in agreement with the extent syntax of subtracting the total length (38 milliseconds) of the preceding durations in the IRstream from the time period (63 milliseconds) of the extent and treating the extent as a gap duration with the value of that time difference.

7.4 Formal syntax

```
bitspec = "<", bare irstream, {"|", bare irstream}, ">";
```

7.5 Semantics

A bitspec is always followed by an irstream. It acts on the contents of that irstream after any bitfield items within the scope of the bitspec have been translated into bit sequences in accordance with 6.3 and any consecutive bit sequences so produced have been concatenated. Its action is to translate such concatenated bit sequences into bare irstreams.

A bitfield itself defines a mapping of time-ordered bit sequences to bare irstreams. If the number of bare irstreams in the bitspec is 2 to the power *n* then these correspond, in their order of occurrence in the bitspec, to the bit sequences given by the bitfields 0:*n*, 1:*n*, 2:*n*, 3:*n*, ... (*n*-1):*n*. These bitfields are translated into time ordered bit sequences in the context of the global sequencing rule.

If the number of bare irstreams in the bitspec is not a power of 2 then *n* is taken as the power in the next greater non-zero power of two and the correspondence between bit sequences and bare irstreams is made in left to right order until the bare irstreams are exhausted. The remaining bit sequences are taken as mapped to empty bare irstreams.

The action of a bitfield on a fully concatenated bit sequence is given by taking its bits in groups of *n* in their time-sequence order starting from the earliest bit and concatenating the bare irstreams that result from mapping each such group. If the number of bits in the bit sequence is not a multiple of *n* then it is a semantic error.

8 Repeat markers

8.1 Description

A repeat marker is an optional element of an IRstream, as described in 6.1, that specifies whether and how execution of that IRstream is to be repeated. The possible repeat markers are:

- "*" which indicates zero or more times;
- "+" which indicates one or more times;
- a number, e.g "3", which indicates that number of times;
- a number followed by "+", which indicates at least that number of times.

So "*" is equivalent to "0+" and "+" to "1+". Assignments and variations in an IRstream may mean that the repeat executions may not all produce the same signal. For repeat markers that specify only a minimum number of repeats, repeats continue beyond the minimum number for as long as the button on the remote is held down, or for one more repeat beyond this if so determined by a variation in accordance with section 12 below. In the execution model of section 14 the remote is a conceptual one.

A simple example is the Dish Network protocol whose inner IRstream was considered in 6.1 In full the protocol is

{57.6k,400}<1,-7|1,-4>(1,-15,(F:-6,U:5,D:5,1,-15)+)

The initial durations 1, -15 are sent only once but the following inner IRstream is sent for as long as the button is held down, but at least once. All repeats are identical, since the inner IRstream contains no assignments or variations.

8.2 Formal syntax

```
repeat marker = "*" | "+" | number, ["+"];
```

8.3 Semantics

A repeat marker following an IRstream specifies whether and how execution of that IRstream is to be repeated. The number of repetitions is determined by the repeat marker as follows: "*" signifies that the IRstream is executed zero or more times, "+" signifies one or more times, a number signifies that many times and a number followed by "+" signifies at least that many times. For repeat markers that specify only a minimum number of times of execution, repeats continue beyond the minimum number for as long as the button on the remote is held down, or for one more repeat beyond this if so determined by the semantics of a variation, in accordance with 12.3 below.

9. Expressions

9.1 Description

An expression signifies a value that is derived from other values by calculation. It consists of names, numbers and bitfields connected by standard symbols for arithmetic and logical operations, the whole being enclosed in parentheses. Bitfields in expressions evaluate to the value of the binary form to which they translate, as described in 1.4, rather than the binary form itself.

Operators have their usual order of precedence and parentheses may be used inside an expression to change the order of precedence by grouping. The permitted operators and rules for evaluation are described in 9.3 below for completeness. Since numbers in IRP notation are purely sequences of digits, explicit negative values have to be represented by an expression in which the operator is the unary minus sign, e.g. (-12).

In the execution model, names that occur in an expression take the values that are bound to them in the global environment at the time that the expression is evaluated. The significance of this is that assignments within an IRstream can change the bindings during the course of execution.

A simple example of the use of an expression is provided by the Panasonic protocol

{37k,432}<1,-1|1,-3>(8,-4, 2:8,32:8,D:8,S:8,F:8,(D^S^F):8,1,-173)+

in which (D^S^F) is an element of a bitfield and denotes the bitwise exclusive-or of the values of the device code D, subdevice code S and function code F.

When a colon appears in an expression, it is a separator in a bitfield. A minus sign that follows a colon is a prefix to the second primary item of a bitfield, as described in 5.1. A minus sign that is not preceded by a primary item must be a unary minus or a syntactic error. All other minus signs are binary operators, again unless they are a syntactic error. Note that a minus sign following a bitfield is preceded by a primary item, namely the final primary item of the bitfield.

Some examples of unusual but valid expressions are:

- (~A:-B:C), the value of the bitfield ~A:-B:C
- (-A:-B:C), unary minus followed by the bitfield A:-B:C
- (A - - 1), binary minus followed by unary minus, equivalent to (A + 1)
- (A:B-C:D), difference of the values of the bitfields A:B and C:D
- (A:(B-C):D), value of a single bitfield with the expression (B-C) as middle primary item
- (A:(B-C:D)), value of a bitfield whose second primary item is the difference of B and the value of the bitfield C:D.

9.2 Formal syntax

```
expression = "(", bare expression, ")";
bare expression = inclusive or expression;
inclusive or expression = [inclusive or expression, "|"], exclusive or
   expression;
exclusive or expression = [exclusive or expression, "^"], and expression;
and expression = [and expression, "&"], additive expression;
additive expression = [additive expression, ("+" | "-")], multiplicative
   expression;
multiplicative expression = [multiplicative expression, ("*" | "/" | "%")],
   exponential expression;
exponential expression = [exponential expression, "**"], unary expression;
unary expression = ["-"], (primary item | bitfield);
(* "primary item" is defined in 5.2 above *)
```

9.3 Semantics

An expression is a type of primary item that determines an integer value through arithmetic and logical operations performed on the values of primary items or bitfields. Within an expression the value of a bitfield is given by translating it into a binary form in accordance with 5.3 and then taking the value of that binary form considered as an integer in binary notation. This value will be a non-negative integer unless the binary form is of infinite length with all leading bits beyond a cartain point being ones, in which case it is interpreted as a negative integer in 1's complement form.

The binary operations "+", "-", "*", "/" denote the arithmetic operations of addition, subtraction, multiplication and division of integers and "%" denotes the modulo (remainder) operation. The division and modulo operations *a/b* and *a%b* are only defined when *b* is nonzero and they are defined such that the inequalities and equation

$$0 <= a\%b < \text{abs}(b), \quad a = (a/b)*b + a\%b$$

are always true, where abs denotes absolute value. This means that in all cases, *a/b* is the greatest integer that is not greater than the algebraic quotient.

The binary operation "**" denotes exponentiation, i.e. *a***b* denotes *a* raised to the power *b*. The unary operation "-" denotes change of sign, i.e. multiplication by -1.

The operations "&", "|", "^" denote the logical operations AND, OR and EXCLUSIVE OR performed bitwise on the binary representations of the values of the operands, where the binary representation of a

negative integer is its usual 1's complement form (but with a potentially infinite number of bits, i.e. with enough bits such that using more bits does not give a different answer). As examples:

(-4)^1 = -5, (-4)^(-1) = 3.

Note: John Fine's original postings say explicitly that he wants to keep the C and C++ shift operators ">>" and "<<" out of the expression syntax. He points out that X<<3 is equivalent to X*8 and X>>3 to the bitfield X::3. With integer division as defined above, X>>3 is also equivalent to X/8, a better parallel with X*8, regardless of whether X is positive or negative.

John does not otherwise say what operators *should* be allowed in expressions but the above seems a reasonable list judging by his examples. I have included exponentiation as it permits an equivalent of a variable shift, since X<<Y and X>>Y are equivalent to X*2**Y and X/2**Y (as "**" has a higher precedence than "*" and "/"). The bitfield X::Y can be used instead of X>>Y but there is no other equivalent of X<<Y.

Evaluation of expressions should be performed in steps with each step being the action of the appropriate binary operator on two values (or the unary minus on one value) in accordance with the parsing syntax. This ensures correct operator precedence since the precedence of the operators, and left-to-right evaluation of expressions such as (x – y – z) that have multiple operators of equal precedence, is incorporated into the formal syntax. For reference, the order of precedence is the usual one. In decreasing order it is:

unary –
**
*, /, %
+, –
&
^
|

There is only one environment in IRP notation, the global environment, but assignments may change the value assigned to a name during the course of execution of the protocol. Expressions are evaluated in the context of the environment as it is at the point of execution of the expression in the execution model.

10. Definitions

10.1 Description

Definitions occur only in the Definitions section of a protocol, which consists of a list of individual definitions separated by commas and enclosed in curly brackets. A *definition* uses an equals sign to bind a bare expression, i.e. an expression without the enclosing parentheses, to a name as a shorthand form that can be used elsewhere in the protocol, including in other definitions. A definition is never evaluated as such. In the execution process its expression is evaluated at the time that the name is evaluated, in the context of the global environment at that time.

Definitions are commonly used to express the calculation of a checksum. A simple example is the Somfy protocol:

{35.7k}<308,-881|669,-520>(2072,-484,F:2,D:3,C:4,-2300)+{C=F*4+D+3}

The plus sign before the definitions section is the repeat marker for the preceding IRstream. This is equivalent to

{35.7k}<308,-881|669,-520>(2072,-484,F:2,D:3,(F*4+D+3):4,-2300)+

A somewhat more complicated example is the DirecTV protocol:

{38k,600,msb}<1,-1|1,-2|2,-1|2,-2>(5,(5,-2,D:4,F:8,C:4,1,-50)+) {C=7*(F:2:6)+5*(F:2:4)
+3*(F:2:2)+(F:2)}

This also shows the use of bitfields in expressions. Enclosing the bitfields in parentheses, as here, is an aid to readability but it is not required by the syntax, which is valid and unambiguous if the parentheses are removed. The Somfy protocol remains easily understood when the expression is substituted for the defined name. The expression is sufficiently complicated in the DirecTV protocol that it would become unwieldy to express the protocol without the use of a definition for the checksum.

Whenever the name occurs as a primary item, as does the C in the IRstream of this example, the expression (with its enclosing parentheses) could be substituted for the defined name so the definition is merely a convenience. One case, however, where a definition provides the only way of using an expression is in a duration or extent. Durations and extents both allow the use of names or numbers but not expressions, so to use an expression it must be bound to a name and that name used in its place.

The definitions list in the Definitions section is read and executed from left to right. This is significant only if two different bare expressions are assigned to the same name. There is no valid reason to do so, but this rule eliminates the ambiguity of which one takes precedence. The right-most one takes precedence as the execution of a definition will supersede any existing binding of the defined name.

Definitions must not be self-referential, either directly or indirectly, as this leads to infinite recursion in the processing of definitions in the execution model. Two examples in which this happens are

{X=F+X} and {X=F+Y, Y=X+D}

The problem it causes is easily seen by repeatedly substituting the corresponding expression for each defined name, as discussed above. Neither the syntax nor the execution model prevents this error, but it would lead to an infinite loop during execution. This is an issue for the protocol writer, just as any programming language can give rise to infinite loops in a badly written program.

10.2 Formal syntax

```
definitions = "{", definitions list, "}";
definitions list = | definition, [",", definitions list - ];
(* This allows a definitions list to be empty but the definitions list in
   the second alternative cannot be empty.  This is the significance of the
   minus sign, which is the EBNF exception symbol.  What is excluded by
   the exception is what follows it, which in this case is empty. *)
definition = name, "=", bare expression;
```

10.3 Semantics

The definitions in a definitions list are executed from left to right. Execution of a definition creates a binding in the global environment. It binds the name on the left of the equals sign to the bare expression on the right, superseding any existing binding of that name.

11. Assignments

11.1 Description

Assignments, like definitions, change the binding of names during the course of execution of a protocol and they have exactly the same form. The distinction is that a definition binds a name to an unevaluated expression, an assignment binds it to the value of an expression. They are distinguished from one another by context. Assignments occur only within IRstreams, definitions only within the Definitions section of a protocol.

When an assignment is executed, the bare expression is evaluated in the global environment as it is at that time. The resulting value is then bound to the name on the left of the assignment, so changing the environment. This makes an assignment such as

```
T = T + 1
```

valid. It increments the value bound to T. Because it is self-referential, this would not be valid as a definition.

Assignments are often used in conjunction with definitions, such as to be able to use the same checksum formula in different circumstances within the same protocol. An example is the OrtekMCE protocol, which can be written as

```
{38.6k,480}<1,-1|-1,1>((P=0,4,-1,D:5,P:2,F:6,C:4,-48m), (P=1,4,-1,D:5,P:2,F:6,C:4,-48m)+,
    (P=2,4,-1,D:5,P:2,F:6,C:4,-48m){C=3+D:1+D:1:1+D:1:2+D:1:3+D:1:4+P:1+P:1:1+
    F:1+F:1:1+F:1:2+F:1:3+F:1:4+F:1:5}
```

The checksum C is three more than the sum of the binary digits in D, P and F together, or put more simply, is three more than the number of "1" bits in their values. It is quite a complicated expression and is used in each of the three IRstreams of the protocol. It can be considered, alternatively, as one IRstream in which the value of P is 0 on the first transmission, 1 on all others except the last one and 2 on the last transmission, which is sent after the button has been released. But it is not only P that changes, so also does the value of the checksum C since this involves P. The use of an assignment means that the value of P in the global environment changes in the course of execution of the protocol, and as C is recalculated from its defined expression each time it is referenced, the correct value is transmitted each time.

Another use of assignments is to represent a toggle that keeps a fixed value throughout the signal from each button press but which toggles between one button press and the next. An example of this behaviour is the RC5 protocol:

```
{36k,msb,889}<1,-1|-1,1>(T=T+1,(1:1,~F:1:6,T:1,D:5,F:6,^114m)+).
```

Only two values are set by a button press, the device code D and function code F. The value of T is maintained by the remote control, incremented on each button press and used to generate the toggle bit T:1 in the IRstream. This is an example of persistence of the global environment, a concept that forms part of the execution model of section 14. The internal state of the remote changes when a button is pressed, and the new state persists until the next button press changes it again. This is modelled by having bindings in the global environment remain in existence until explicitly changed by the pressing of a button on the conceptual remote of the execution model. The value of T is unknown when the protocol is executed, but whatever it may be, it will be one greater on the next button press.

As a final example of the use of assignments, consider the AirAsync protocol:

(37.7k,840)<1|-1>(N=0,(1,B:8:N,-2,N=N+8)+)

The value of N is 0 on the first transmission of the repeating IRstream but increases by 8 on each successive transmission. The bitfield B:8:N is therefore the least significant 8 bits on the first transmission, the next 8 on the next transmission, and so on. The sequencing rule is lsb, so the result is that the value of B is sent in lsb order, eight bits at a time, on successive repeats, however large may be the value of B. This protocol represents standard asynchronous transmission with one start bit, eight data bits and two stop bits.

11.2 Formal syntax

```
assignment = name, "=", bare expression;
```

11.3 Semantics

When an assignment is executed, the bare expression is evaluated according to the current state of the global environment. The resulting value is then bound to the given name, so changing the environment.

12. Variations

12.1 Description

The OrtekMCE protocol considered in 11.1 had an IRstream that varied systematically between its first transmission, its last one and all others in between. The first one is sent on button down, the middle ones while the button remains held down and the last one on button up. A number of other protocols share this pattern, yet more share a pattern of variation between button down and button held but without a transmission on button up.

These patterns can be represented in IRP notation without the need for any new structure, as the example of the OrtekMCE protocol showed. When the protocol is written out in full in the form given in 11.1, it does however obscure the fact that the three different IRstreams are all small variations of one another. The use in IRstreams of a *variation* both simplifies the appearance of the protocol and clarifies this relationship between the various transmissions.

A *variation* consists of two or three alternative bare IRstreams given together, each enclosed in square brackets as delimiters. Typically the bare IRstreams will consist only of a single item, but this is not required. On first transmission (button down) the first alternative is used, on middle transmissions (button held) the second alternative is used, and if a third alternative is present then on button up the

IRstream is sent once more with the third alternative being used. The OrtekMCE protocol can therefore be written in the simpler form

{38.6k,480}<1,-1|-1,1>([P=0][P=1][P=2],4,-1,D:5,P:2,F:6,C:4,-48m)+{C=3+D:1+D:1:1+
D:1:2+D:1:3+ D:1:4+P:1+P:1:1+F:1+F:1:1+F:1:2+F:1:3+F:1:4+F:1:5}

A simpler example is the CanalSat protocol:

{55.5k,250,msb}<-1,1|1,-1>([T=0][T=2],1:1,D:7,S:6,T:2,F:7,-89m)+

Here there is no dependent checksum, the protocol could be written without the use of T as

{55.5k,250,msb}<-1,1|1,-1>((1:1,D:7,S:6,0:2,F:7,-89m), (1:1,D:7,S:6,2:2,F:7,-89m)+)

but use of a variation emphasises that there is only a small systematic difference between the first and subsequent frames.

There are a few protocols in which one IRstream is transmitted on button down, a related one on button up, but nothing while the button is held. This typically corresponds to buttons of a mouse, where holding down a mouse button does nothing but there is an effect on button up. To meet this situation, an alternative with an empty IRstream signifies that the remainder of the IRstream should be skipped. An example is the variant of the Sejin protocol that corresponds to the buttons of a three-button mouse. This is:

{38.8k,310,msb}<-1|1>(<8:4|4:4|2:4|1:4>([F=B][ ][F=0],3,3:2,(-D):6,B:2,0:16,E:4,C:4,-3600u)+)
{C=((-D):4:2+4*(-D):2+F+E)&15}

In this, D is the device code but its negative is sent when the mouse part of the protocol is used. B numbers the buttons, 1,2 or 3. On button up the IRstream is sent with B=0, the same for all buttons as the button down part has shown which button is concerned. The checksum C is calculated with either the button number or 0 as appropriate. The middle alternative [ ] indicates that nothing is sent while the button is held.

12.2 Formal syntax

```
variation = 2*alternative, [alternative];
alternative = "[", bare irstream, "]";
```

12.3 Semantics

A variation is a permitted type of irstream item that operates in conjunction with a repeat marker for the irstream that contains it. If the repeat marker specifies an indefinite number of times of execution with a given minimum then the state, held or released, of the button of the remote control affects the effect of the variation but this is not so if the number of repeats is definite.

With a definite number in the repeat marker, on the first execution of the IRstream the bare IRstream of any first alternative is executed. If a variation is encountered with three alternatives then on the last execution of the IRstream, provided this is not also the first, that of the third alternative is executed. In all other cases that of the second alternative is executed.

The same holds with an indefinite number in the repeat marker but an additional rule is required to determine the actual number of times that the IRstream is executed. The IRstream is repeatedly executed for as long as the button is held down at the start of an execution. Repeats also continue beyond that, if necessary, until all of the following are true:

- the minimum number given in the repeat marker has been performed;
- at the end of the last execution the button is no longer held;
- the successive executions satisfy the rule for a fixed number of repeats with that fixed number being the actual number.

Note the implicit nature of these conditions. It is not possible to tell whether an execution is going to be the last one if the button was still held when it commenced, so if a variation with three alternatives is encountered, it may be necessary to perform one further execution after the button has been released in order to meet the third condition.

If the bare irstream of an alternative is empty then its execution consists of skipping the remainder of the IRstream and proceeding to its next execution.

13. Names and numbers

13.1 Description

For the sake of completeness, names and numbers are defined formally in this section. They were defined informally in 1.2. All that is needed to add here is that the presence of assignments or variations may mean that a name may not evaluate to the same value on each occasion that it is evaluated. A name bound to a value evaluates to the value that is bound to it in the global environment at the time that it is evaluated. A name bound to a bare expression is evaluated by evaluating that expression.

13.2 Formal syntax

```
number = "0" | nonzero digit, {digit};
number with decimals = number, [".", digit, {digit} ];
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
nonzero digit = digit - "0";
name = letter, {letter | digit};
letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"
   |"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z";
```

13.3 Semantics

The value of a number is the non-negative integer represented by its sequence of digits in decimal notation.

The value of a number with decimals is the non-negative real number represented by its sequence of digits and optional decimal point in decimal notation.

The value of a name is the value bound to it at the time of evaluation, or is the result of evaluating the expression bound to it at that time, as the case may be.

14. Execution model

14.1 Introduction

The level of recursion permitted in the IRP notation is sufficiently great that a formal execution model is needed to ensure a consistent interpretation of some of its more complex possibilities. This section provides such a model in a natural language formulation.

The execution environment conceptually contains a timer and a carrier generator. All processes are considered as instantaneous other than TRANSMIT, which turns on the carrier generator for a specified time, and WAIT, which waits for a specified time. Processes are synchronous, so that any process that follows a TRANSMIT or WAIT is not performed until the TRANSMIT or WAIT is completed.

The execution model has two aspects, static and dynamic. The static aspect describes such things as how expressions are interpreted and how bitfields are converted to binary forms. Collectively the operations of the static aspect are described as evaluation. They are performed by the operator EVALUATE. The dynamic aspect describes such things as how IRstreams are processed to produce the flashes and gaps of the signal. The dynamic procedures are performed by the operator PROCESS. Both aspects are covered by the semantics specified in the previous sections but this execution model puts them together to form a coherent whole.

Processing is initiated by the pressing of a button on a conceptual remote control. The button press binds certain names to values in the global environment. These correspond to the Device Code and OBC or the like, the values that are set externally by means outside the scope of the IRP notation. Also set externally is the duty cycle of the carrier generator. This is typically 33% or thereabouts, but again it is outside the scope of the IRP notation. With these external processes performed, the button press initiates the procedure "PROCESS protocol".

The global environment is considered to have a continuing existence between button presses. The external process of binding names to values when a button is pressed changes the binding of those names but all other names are considered to have bound values as well, albeit possibly unknown values. This feature of the model permits names to be used that are neither bound to a value by a button press nor bound to an expression by a definition. This is the mechanism by which certain types of toggle can be represented. Some protocols have a toggle bit that alternates in value between successive button pressings, to enable successive presses of the same button to be distinguished from a single longer press. When a user picks up the remote and presses a button, the state of the toggle that will be used is unknown. What is known is that it will be different on the next press. The remote has "state", i.e. it maintains internal variables such that the same user action at different times may not result in precisely the same effect. Persistence of the global environment permits modelling of remotes with "state".

14.2 Data structures

The dynamic aspect of the execution model processes the main bitspec and irstream of a protocol in a context set by the timer and carrier generator and certain parameters of fixed global scope. Since there can be further bitspecs and irstreams nested within the main irstream, the model uses stacks to keep track of which ones are current at any stage of the processing. The following data structures are used to model these various items.

```
STRUCT timer
BEGIN
    clock:          real-time clock counting microseconds;
    now:            the current time on the clock;
    START:          operation that sets clock to zero and starts it running;
    STOP:           operation that stops the clock;
END

STRUCT carrier generator
BEGIN
    frequency:      the frequency in kilohertz of the carrier square wave;
    duty cycle:     the duty cycle of the carrier square wave;
    TRANSMIT FOR:   an operation that turns on the carrier for a specified time;
    WAIT FOR:       an operation that waits with the carrier turned off for a specified time;
END

STRUCT global
BEGIN
    sequencing rule:  an enumerated value, either "lsb" or "msb", that specifies the manner in which
                      binary forms are converted to bit sequences;
    duration unit:    the unit of time in microseconds in which durations and extents without a suffix are
                      expressed;
END

STRUCT bitspec
BEGIN
    (* The bitspec buffer performs the process of converting binary forms to bit sequences *)
    N:              the length of the bit sequences mapped by the bitspec;
    buffer:         a FIFO buffer of bits awaiting processing by the bitspec, initially empty; PUSH
                    pushes a binary form into the buffer one bit at a time (conceptually into the right
                    hand end) in the order given by the global sequencing rule, PULL pulls a time-
                    ordered bit sequence of N bits out one at a time (conceptually out of the left hand
                    end), earliest bit first;
    MAP:            operation that maps bit sequences of length N to their corresponding bare irstreams;
END

STRUCT irstream
BEGIN
    content:        the bare irstream  (a list of irstream items)
    min count:      the minimum number of times that the irstream should be executed, as specified by
                    the repeat marker — in the case of a definite repeat number this is the actual number;
    repeat flag:    boolean that is false if the repeat marker gives a definite number and is true if the
                    repeat marker gives only a minimum number;
    bitspec flag:   boolean specifying whether or not the irstream immediately follows a bitspec, set
                    during parsing;
    final flag:     boolean that controls the ending of an indefinite repeat sequence, tested at the start of
                    a potential repeat execution it the button is then no longer held — if this is false then
                    that repeat is not performed, if it is true then that repeat is performed;
    var3 flag:      boolean that is set true during execution of an irstream if during that execution a
                    variation is encountered that has three alternatives — used in the setting of the final
                    flag;
```

```
    skip flag:      boolean specifying whether processing of the remainder of this irstream should be
                    skipped, initially false — used in the processing of an empty alternative;
    count:          the number of transmissions currently made or in progress, initially zero
    start time:     a time value set when processing of the irstream starts
END

STRUCT <item> stack
BEGIN
    stack:          an ordered list of <item>, conceptually thought of as climbing upward
    pointer:        a pointer to a (possibly empty) position in the stack; INCREMENT moves the pointer
                    up one position, DECREMENT moves it down one position
    CREATE:         an operation that creates an empty stack of <item> with pointer set to first (bottom)
                    position;
    DESTROY:        an operation that deletes a stack, empty or otherwise;
    INSERT INTO:    a stack operation in which the <item>s at and beyond the pointer position are moved
                    up one place, a specified new item is inserted at the pointer position and the pointer is
                    then incremented;
    DELETE FROM:    a stack operation in which the pointer is decremented, the <item> at the pointer
                    position is deleted and all <item>s beyond the pointer position are moved down one
                    place;
    current:        the <item> one position lower than the pointer position (null if the pointer is at the
                    bottom of the stack);
    (* In normal circumstances <item>s will only be added and removed at the top of the stack, in which
    case INSERT INTO and DELETE FROM correspond to PUSH and PULL.  Unless there have been
    explicit pointer increments or decrements the current <item> will then be the one at the top of the
    stack.  The recursion allowed by IRP notation can however cause <item>s to be inserted part way up.
    *)
END
```

14.3 Evaluation

The static operation of EVALUATE was described in 14.1 Although evaluation is static in the sense that it is not concerned with the progress of time during execution, evaluations are perfomed in the context of the state of the environment at the time that the EVALUATE operation is called. Successive evaluations of the same item may therefore give different results.

Although the process of evaluation is described by the semantics of the item being evaluated, descriptions are given here for each item type on which EVALUATE can act.

```
EVALUATE number
BEGIN
    RETURN value of the number as an integer in accordance with number semantics;
END

EVALUATE name
BEGIN
    IF name is bound to a value THEN
        RETURN value;
    ELSE IF name is bound to an expression THEN
        EVALUATE expression AS value;
        RETURN value;
```

```
    END IF;
    (* A name that has not been bound explicitly will still return a value.  The description of the
    persistence of the global environment in 14.1 explains how this can be used to model remotes with
    "state" where this internal state affects the execution of the protocol. *)
END

EVALUATE expression
    Parse the expression in accordance with expression syntax;
    EVALUATE each primary item in the parse tree according to its type;
    Convert any binary form arising from a bitfield to its numerical value in accordance with expression
    semantics;
    RETURN value given by performing the unary and binary operations in the parse tree in accordance
    with expression semantics;
END

EVALUATE duration
BEGIN
    EVALUATE its name or number;
    Convert value to a time period in microseconds in accordance with the duration suffix and the
    duration semantics in the context of the global duration unit and carrier generator frequency;
    RETURN time period in microseconds;
END

EVALUATE extent
BEGIN
    EVALUATE its name or number;
    Convert value to a time period in microseconds in accordance with the extent suffix and the extent
    semantics in the context of the global duration unit and carrier generator frequency;
    RETURN time period in microseconds;
END

EVALUATE bitfield
BEGIN
    EVALUATE each primary item according to its type;
    Convert bitfield to binary form in accordance with bitfield semantics;
    RETURN binary form
END
```

14.4 Execution

As described in 14.1, execution processing is initiated by the pressing of a button on the conceptual remote control. By means outside the scope of IRP notation, this binds certain names to values in the global environment and sets the carrier generator duty cycle. It then initiates the procedure *PROCESS protocol*. This section describes the chain of processes that this initiates.

Processing of item types is specified in terms of their parsing by the EBNF syntax and the data structures that represent them in accordance with 14.2. In logical constructions, AND takes precedence over OR. The pronoun ITS is used to identify a part of the parsing structure or data structure of the item being processed, OF is used to identify a part of a different item. For example, the EBNF syntax for a bare irstream is

```
bare irstream = | irstream item, [",", bare irstream];
```

So in "PROCESS bare irstream", "IF bare irstream IS NOT EMPTY" refers to the bare irstream being processed (the one on the left of the syntax expression) but "PROCESS ITS bare irstream" refers to the one on the right. In the same context, "PROCESS buffer OF current bitspec" refers to a constituent of the data structure that represents the current bitspec, not to one of the bare irstream being processed. Comments have been made where there is potential ambiguity.

The processes of the execution model now follow.

```
PROCESS protocol
BEGIN
    CREATE bitspec stack;
    CREATE irstream stack;
    PROCESS ITS generalspec;
    PROCESS ITS definitions;
    START clock;
    PROCESS ITS bitspec;
    PROCESS ITS irstream;
    STOP clock;
    (* The binding of names to values is preserved in the persistence of the environment but the binding
    of names to bare expressions produced by the processing of definitions is not preserved. *)
    CLEAR ITS definitions;
    DESTROY irstream stack;
    DESTROY bitspec stack;
END

PROCESS generalspec
BEGIN
    SET carrier generator frequency FROM ITS frequency item;
    SET global duration unit FROM ITS unit item;
    SET global sequencing rule FROM ITS order item;
END

PROCESS definitions
BEGIN
    PROCESS ITS definitions list;
END

PROCESS definitions list
BEGIN
    IF definitions list IS NOT EMPTY THEN
        PROCESS ITS (* first *) definition;
        UNLESS ABSENT PROCESS ITS (* remaining *) definitions list;
    END IF
END

PROCESS definition
BEGIN
    BIND ITS name TO ITS bare expression;
```

```
END

PROCESS bitspec
BEGIN
    INSERT bitspec INTO bitspec stack;
END

PROCESS irstream
BEGIN
    INSERT irstream INTO irstream stack;
    SET ITS final flag = FALSE;
    SET ITS count = 0;
    DO WHILE ITS count < ITS min count OR (ITS repeat flag IS TRUE AND
                        (button still held OR ITS final flag IS TRUE))
        SET ITS var3 flag = FALSE;
        SET ITS skip flag = FALSE;
        INCREMENT ITS count;
        SET ITS start time = now;
        (* Evaluate content in current environment settings *)
        PROCESS ITS bare irstream;
        (* Treat end of bare irstream like item that is not a bitfield *)
        PROCESS buffer OF current bitspec;
        IF NOT (ITS repeat flag IS TRUE AND button still held)
                (* this could never be known in advance to be final *)
            IF (ITS var3 flag IS TRUE OR ITS count EQUALS (ITS min count – 1) )
                        AND ITS final flag IS FALSE THEN
                (* next execution needed and known to be final  *)
                SET ITS final flag = TRUE;
            ELSE
                (* current execution was final, whether this was known when it started or not *)
                SET ITS final flag = FALSE;
            END IF;
        END IF;
    END DO;
    IF ITS bitspec flag IS TRUE THEN
        (* We have finished with the bitspec that preceded this irstream *)
        DELETE FROM bitspec stack;
    END IF
    DELETE FROM irstream stack;
END

PROCESS bare irstream
BEGIN
    IF skip flag OF current irstream IS FALSE AND bare irstream IS NOT EMPTY THEN
        (* Process first item *)
        IF ITS irstream item IS NOT bitfield THEN
            PROCESS buffer OF current bitspec;
        END IF;
        PROCESS ITS irstream item;
        (* Process the remaining items — if skip flag has just been set then this will be skipped due to
        the conditional at the start of this process *)
```

```
        UNLESS ABSENT PROCESS ITS bare irstream;
    END IF
    (* If the remaining bare irstream is absent, so we have completed the processing, it seems we should
    process bitspec buffer, treating the end of the recursion as an item other than a bare irstream, but this
    would be wrong as another bare irstream may follow, starting with a bitfield whose bits need to be
    concatenated with any left over here.  We defer any final processing of the bitspec buffer till the end
    of the enclosing irstream. *)
END

PROCESS irstream item
BEGIN
    IF irstream item IS duration THEN
        EVALUATE duration AS time period;
        IF duration IS flash duration THEN
            TRANSMIT FOR time period;
        ELSE (* duration is gap duration *)
            WAIT FOR time period;
        END IF;
    ELSE IF irstream item IS extent THEN
        EVALUATE extent AS time period;
        IF now < (start time OF current irstream) + time period THEN
            WAIT UNTIL now =  (start time OF current irstream) + time period;
        END IF;
    ELSE IF irstream item IS bitfield THEN
        EVALUATE bitfield AS binary form;
        IF current bitspec IS NULL THEN
            ERROR;
        ELSE
            PUSH binary form INTO buffer OF current bitspec;
        END IF;
    ELSE IF irstream item IS assignment THEN
        EVALUATE ITS expression AS value;
        BIND ITS name TO value;
    ELSE IF irstream item IS variation THEN
        IF number of alternatives in variation IS 3 THEN
            SET var3 flag OF current irstream = TRUE;
        END IF;
        IF count OF current irstream IS 1 THEN
            PROCESS ITS first alternative;
        ELSE IF ITS third alternative IS PRESENT AND final flag OF current irstream IS TRUE THEN
            PROCESS ITS third alternative;
        ELSE
            PROCESS second alternative;
        END IF;
    ELSE IF irstream item IS bitspec THEN
        PROCESS irstream item AS bitspec;
    ELSE IF irstream item IS irstream THEN
         PROCESS irstream item AS irstream;
    END IF;
END
```

```
PROCESS alternative
BEGIN
    IF ITS bare irstream IS EMPTY THEN
        SET skip flag OF current irstream = TRUE;
    ELSE
        PROCESS ITS bare irstream
    END IF
END

PROCESS buffer
BEGIN
    (* This buffer is in scope of previous bitspec, if any *)
    IF buffer IS NOT EMPTY THEN
        DECREMENT bitspec stack pointer;
        DO WHILE number of bits in buffer >= N
            PULL bit sequence FROM buffer;
            MAP bit sequence TO bare irstream;
            PROCESS bare irstream;
        END DO;
        (* The current bitspec now was the previous bitspec on entry *)
        IF current bitspec IS NOT NULL THEN
            (* if an attempt has been made to push bits into its buffer, it will already have
            generated an error *)
            PROCESS buffer OF current bitspec;
        END IF;
        IF number of bits in buffer <> 0 THEN
            (* Consecutive bit sequences have been concatenated by processing of bare irstream
                so if any bits are left over, it is an error. *)
            ERROR;
        END IF;
        (* Finished with the new current bitspec, restore the old one *)
        INCREMENT bitspec stack pointer;
    END IF;
END;


CLEAR definitions
BEGIN
    CLEAR ITS definitions list;
END

CLEAR definitions list
BEGIN
    IF definitions list IS NOT EMPTY THEN
        CLEAR ITS (* first *) definition;
        UNLESS ABSENT CLEAR ITS (* remaining *) definitions list;
    END IF;
END
```

```
CLEAR definition
BEGIN
    BIND ITS name TO zero;
    (* Definitions are not preserved in the persistence of the global environment so clear the
    binding of the name to a bare expression and bind it instead to a value.  There is no particular
    significance in the value zero. *)
END
```

Graham Dixon (based on work of John Fine)
4 February 2010