```java
/*
 * Copyright (C) 2016 redxef.
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301  USA
 */
package powerspy.baselib;

import java.nio.ByteBuffer;
import java.util.Arrays;
import static powerspy.baselib.IODefs.*;

/**
 *
 * @author redxef
 */
public class DataPacket {

        private static final byte INT8_BYTES = 1;
        private static final byte INT16_BYTES = 2;
        private static final byte INT24_BYTES = 3;
        private static final byte INT32_BYTES = 4;
        private static final byte FLOAT_BYTES = 3;

        private char type;
        private int length;
        private final byte[] data;
        private boolean string_fin;

        private int position;

        /**
         * Creates a new, empty DataPacket with BUFFER_SIZE data buffer.
         */
        public DataPacket()
        {
                type = NONE;
                length = 0;
                data = new byte[BUFFER_SIZE];
                position = 0;
                string_fin = false;
        }

        /**
```

```
57          * Sets the type of this DataPacket. Every value is valid, but only the
58          * ones defined in IODefs are of real use.
59          *
60          * @param c the type
61          */
62         public void setType(char c)
63         {
64                 type = c;
65         }
66
67         /**
68          * Adds a Byte of data to the end of the data[] and increments the write
69          * counter.
70          *
71          * @param b the data to write
72          */
73         public void addByte(byte b)
74         {
75                 if (!isFinished()) {
76                         data[position++] = b;
77                         length++;
78                 }
79         }
80
81         /**
82          * Sets the finished String flag.
83          */
84         public void setFinishedString()
85         {
86                 string_fin = true;
87         }
88
89         /**
90          * Returns the type of this DataPacket.
91          *
92          * @return the type
93          */
94         public char getType()
95         {
96                 return type;
97         }
98
99         /**
100         * Returns the length of this data, this is NOT the length of the
101         * byte[].
102         *
103         * @return the number of bytes in this DataPacket
104         */
105        public int getLength()
106        {
107                return length;
108        }
109
110        /**
111         * Seeks the fornt of this DataPacket.
112         */
113        public void seekFront()
```

```
114         {
115                 position = 0;
116         }
117
118         /**
119          * Reads the next Byte of data from the array and returns it.
120          *
121          * @return the read data
122          *
123          * @throws PackageException when the end of the buffer is accessed
124          */
125         public byte readNext() throws PackageException
126         {
127                 if (position == data.length) {
128                         throw new PackageException("reached end of package");
129                 }
130
131                 return data[position++];
132         }
133
134         /**
135          * Returns the next item in the buffer array. This is the same as
136          * {@code readNext()}.
137          *
138          * @return the read data
139          *
140          * @throws PackageException when the end of the buffer is accessed
141          */
142         public byte getChar() throws PackageException
143         {
144                 return readNext();
145         }
146
147         /**
148          * Returns the number of available bytes.
149          *
150          * @return the number of bytes
151          */
152         public int availableChar()
153         {
154                 return length - position;
155         }
156
157         /**
158          * Returns true if this DataPacket has no byte stored in it, otherwise
159          * false.
160          *
161          * @return if this DataPacket is empty
162          */
163         public boolean isEmpty()
164         {
165                 for (int i = 0; i < data.length; i++) {
166                         if (i != 0)
167                                 return false;
168                 }
169                 return true;
170         }
```

```
171
172         private boolean isFinishedNr()
173         {
174                 switch (type) {
175                         case INT8:
176                         case UINT8:
177                                 return length == INT8_BYTES;
178                         case INT16:
179                         case UINT16:
180                                 return length == INT16_BYTES;
181                         case INT24:
182                         case UINT24:
183                                 return length == INT24_BYTES;
184                         case INT32:
185                         case UINT32:
186                                 return length == INT32_BYTES;
187                         case FLOAT:
188                                 return length == FLOAT_BYTES;
189                         default:
190                                 return false;
191                 }
192         }
193
194         private boolean isFinishedString()
195         {
196                 return string_fin;
197         }
198
199         /**
200          * Checks if this DataPacket is finished, effectively preventing any
201          * further modification.
202          *
203          * @return true if the Packet is finished, otherwise false
204          */
205         public boolean isFinished()
206         {
207                 if (type >= INT8 && type <= UINT32 || type == FLOAT) {
208                         return isFinishedNr();
209                 } else if (type == STRING) {
210                         return isFinishedString();
211                 } else {
212                         return false;
213                 }
214         }
215
216         /**
217          * Checks if this DataPacket is of type INT8.
218          *
219          * @return true if the type matches
220          */
221         public boolean isInt8()
222         {
223                 return getType() == INT8;
224         }
225
226         /**
227          * Checks if this DataPacket is of type INT16.
```

```
228          *
229          * @return true if the type matches
230          */
231         public boolean isInt16()
232         {
233                 return getType() == INT16;
234         }
235
236         /**
237          * Checks if this DataPacket is of type INT24.
238          *
239          * @return true if the type matches
240          */
241         public boolean isInt24()
242         {
243                 return getType() == INT24;
244         }
245
246         /**
247          * Checks if this DataPacket is of type INT32.
248          *
249          * @return true if the type matches
250          */
251         public boolean isInt32()
252         {
253                 return getType() == INT32;
254         }
255
256         /**
257          * Checks if this DataPacket is of type UINT8.
258          *
259          * @return true if the type matches
260          */
261         public boolean isUInt8()
262         {
263                 return getType() == UINT8;
264         }
265
266         /**
267          * Checks if this DataPacket is of type UINT16.
268          *
269          * @return true if the type matches
270          */
271         public boolean isUInt16()
272         {
273                 return getType() == UINT16;
274         }
275
276         /**
277          * Checks if this DataPacket is of type UINT24.
278          *
279          * @return true if the type matches
280          */
281         public boolean isUInt24()
282         {
283                 return getType() == UINT24;
284         }
```

```
285
286         /**
287          * Checks if this DataPacket is of type UINT32.
288          *
289          * @return true if the type matches
290          */
291         public boolean isUInt32()
292         {
293                 return getType() == UINT32;
294         }
295
296         /**
297          * Checks if this DataPacket is of type Float.
298          *
299          * @return true if the type matches
300          */
301         public boolean isFloat()
302         {
303                 return getType() == FLOAT;
304         }
305
306         /**
307          * Checks if this DataPacket is of type String.
308          *
309          * @return true if the type matches
310          */
311         public boolean isString()
312         {
313                 return getType() == STRING;
314         }
315
316         /**
317          * Reads an unsigned 8 bit Integer from the data[].
318          *
319          * @return the Integer
320          *
321          * @throws PackageException if the type of the packet doesn't match with
322          *                          the type to read.
323          */
324         public int readUInt8() throws PackageException
325         {
326                 int res;
327                 byte[] b = new byte[4];
328                 if (!isUInt8()) {
329                         throw new PackageException("Wrong package type.");
330                 }
331
332                 seekFront();
333
334                 b[0] = 0;
335                 b[1] = 0;
336                 b[2] = 0;
337                 b[3] = getChar();
338
339                 res = ByteBuffer.wrap(b).getInt();
340
341                 return res;
```

```
342         }
343
344         /**
345          * Reads an unsigned 16 bit Integer from the data[].
346          *
347          * @return the Integer
348          *
349          * @throws PackageException if the type of the packet doesn't match with
350          *                          the type to read.
351          */
352         public int readUInt16() throws PackageException
353         {
354                 int res;
355                 byte[] b = new byte[4];
356                 if (!isUInt16()) {
357                         throw new PackageException("Wrong package type.");
358                 }
359
360                 seekFront();
361                 b[0] = 0;
362                 b[1] = 0;
363                 b[2] = getChar();
364                 b[3] = getChar();
365
366                 res = ByteBuffer.wrap(b).getInt();
367
368                 return res;
369         }
370
371         /**
372          * Reads an unsigned 24 bit Integer from the data[].
373          *
374          * @return the Integer
375          *
376          * @throws PackageException if the type of the packet doesn't match with
377          *                          the type to read.
378          */
379         public int readUInt24() throws PackageException
380         {
381                 int res;
382                 byte[] b = new byte[4];
383                 if (!isUInt24()) {
384                         throw new PackageException("Wrong package type.");
385                 }
386
387                 seekFront();
388                 b[0] = 0;
389                 b[1] = getChar();
390                 b[2] = getChar();
391                 b[3] = getChar();
392
393                 res = ByteBuffer.wrap(b).getInt();
394
395                 return res;
396         }
397
398         /**
```

```
399          * Reads an unsigned 32 bit Integer from the data[].
400          *
401          * @return the Integer
402          *
403          * @throws PackageException if the type of the packet doesn't match with
404          *                          the type to read.
405          */
406         public int readUInt32() throws PackageException
407         {
408                 int res;
409                 byte[] b = new byte[4];
410                 if (!isUInt32()) {
411                         throw new PackageException("Wrong package type.");
412                 }
413
414                 seekFront();
415                 b[0] = getChar();
416                 b[1] = getChar();
417                 b[2] = getChar();
418                 b[3] = getChar();
419
420                 res = ByteBuffer.wrap(b).getInt();
421
422                 return res;
423         }
424
425         /**
426          * Reads an 8 bit Integer from the data[].
427          *
428          * @return the Integer
429          *
430          * @throws PackageException if the type of the packet doesn't match with
431          *                          the type to read.
432          */
433         public int readInt8() throws PackageException
434         {
435                 int res;
436                 byte[] b = new byte[4];
437                 if (!isInt8()) {
438                         throw new PackageException("Wrong package type.");
439                 }
440
441                 seekFront();
442
443                 b[0] = 0;
444                 b[1] = 0;
445                 b[2] = 0;
446                 b[3] = getChar();
447
448                 res = ByteBuffer.wrap(b).getInt();
449
450                 if ((res & 0x80) > 0) {
451                         res = ~res + 1; //invert
452                         res &= 0xff;
453                         res = ~res + 1;
454                 }
455                 return res;
```

```
456         }
457
458         /**
459          * Reads an 16 bit Integer from the data[].
460          *
461          * @return the Integer
462          *
463          * @throws PackageException if the type of the packet doesn't match with
464          *                          the type to read.
465          */
466         public int readInt16() throws PackageException
467         {
468                 int res;
469                 byte[] b = new byte[4];
470                 if (!isInt16()) {
471                         throw new PackageException("Wrong package type.");
472                 }
473
474                 seekFront();
475
476                 b[0] = 0;
477                 b[1] = 0;
478                 b[2] = getChar();
479                 b[3] = getChar();
480
481                 res = ByteBuffer.wrap(b).getInt();
482
483                 if ((res & 0x8000) > 0) {
484                         res = ~res + 1; //invert
485                         res &= 0xffff;
486                         res = ~res + 1;
487                 }
488                 return res;
489         }
490
491         /**
492          * Reads an 24 bit Integer from the data[].
493          *
494          * @return the Integer
495          *
496          * @throws PackageException if the type of the packet doesn't match with
497          *                          the type to read.
498          */
499         public int readInt24() throws PackageException
500         {
501                 int res;
502                 byte[] b = new byte[4];
503                 if (!isInt24()) {
504                         throw new PackageException("Wrong package type.");
505                 }
506
507                 seekFront();
508
509                 b[0] = 0;
510                 b[1] = getChar();
511                 b[2] = getChar();
512                 b[3] = getChar();
```

```
513
514                 res = ByteBuffer.wrap(b).getInt();
515
516                 if ((res & 0x800000) > 0) {
517                         res = ~res + 1; //invert
518                         res &= 0xffffff;
519                         res = ~res + 1;
520                 }
521                 return res;
522         }
523
524         /**
525          * Reads an 32 bit Integer from the data[].
526          *
527          * @return the Integer
528          *
529          * @throws PackageException if the type of the packet doesn't match with
530          *                          the type to read.
531          */
532         public int readInt32() throws PackageException
533         {
534                 int res;
535                 byte[] b = new byte[4];
536                 if (!isInt32()) {
537                         throw new PackageException("Wrong package type.");
538                 }
539
540                 seekFront();
541
542                 b[0] = getChar();
543                 b[1] = getChar();
544                 b[2] = getChar();
545                 b[3] = getChar();
546
547                 res = ByteBuffer.wrap(b).getInt();
548
549                 if ((res & 0x80000000) > 0) {
550                         res = ~res + 1; //invert
551                         res &= 0xffffffff;
552                         res = ~res + 1;
553                 }
554                 return res;
555         }
556
557         /**
558          * Reads an 24 bit Float from the data[].
559          *
560          * @return the Float
561          *
562          * @throws PackageException if the type of the packet doesn't match with
563          *                          the type to read.
564          */
565         public float readFloat() throws PackageException
566         {
567                 int res;
568                 byte[] b = new byte[4];
569                 if (!isFloat()) {
```

```
570                     throw new PackageException("Wrong package type.");
571             }
572
573             seekFront();
574             b[0] = 0;
575             b[1] = getChar();
576             b[2] = getChar();
577             b[3] = getChar();
578
579             res = ByteBuffer.wrap(b).getInt();
580
581             return Float.intBitsToFloat(res << 8);
582     }
583
584     /**
585      * Reads an String from the data[].
586      *
587      * @return the String
588      *
589      * @throws PackageException if the type of the packet doesn't match with
590      *                          the type to read.
591      */
592     public String readString() throws PackageException
593     {
594             StringBuilder sb = new StringBuilder();
595             if (!isString()) {
596                     throw new PackageException("Wrong package type.");
597             }
598
599             seekFront();
600             while (availableChar() > 0) {
601                     sb.append((char) getChar());
602             }
603
604             return sb.toString();
605     }
606
607     /**
608      * Reads any type of data from the data[]. This is a convenience method
609      * intended for debugging.
610      *
611      * @return the result as Object or null if the Package is not finished
612      *
613      * @throws PackageException this Exception is never thrown
614      */
615     public Object readObj() throws PackageException
616     {
617             switch (getType()) {
618                     case INT8:
619                             return readInt8();
620                     case INT16:
621                             return readInt16();
622                     case INT24:
623                             return readInt24();
624                     case INT32:
625                             return readInt32();
626                     case UINT8:
```

```
627                             return readUInt8();
628                     case UINT16:
629                             return readUInt16();
630                     case UINT24:
631                             return readUInt24();
632                     case UINT32:
633                             return readUInt32();
634                     case FLOAT:
635                             return readFloat();
636                     case STRING:
637                             return readString();
638                     default:
639                             return null;
640             }
641     }
642
643     @Override
644     public String toString()
645     {
646             StringBuilder sb = new StringBuilder();
647
648             sb.append("type: ");
649             sb.append(type);
650             sb.append("; data: ");
651             sb.append(Arrays.toString(data));
652             sb.append(";");
653
654             return sb.toString();
655     }
656 }
```