Arduino errorhandling Library

Garrett Berg, Cloudform Design, garrett@cloudformdesign.com

Loosely based on code from the fantastic "*Learn C the Hard Way*", Zed A. Shaw,
http://c.learncodethehardway.org/book/ex20.html

Summary

This is the Arduino errorhandling library, meant to make debugging and error handling be simple and work together.

Intended Use

There are many times where you want to do error checking and simultaneously provide debugging. This library helps significantly. Errors are automatically printed if DEBUG is defined.

This library offers two ways to deal with errors,:

1. **return checks** — these do all your logging for you and set **errno** (the global error variable). They then return R. These are intended for "simple use" cases, where you do not need to do any error handling (like send special error messages, close ports, etc)

2. **goto error checks** — these do your logging, etc and then "*goto error*" where you can do error handling like closing ports, etc.

In the the calling functions, you can then noerr_log_return() to follow the stack trace down so you can see what called what. This helps you keep track of where your code is giving errors. For example, in “errorhandling_ex1” if you type ’x’ (an invalid input), you will get:

```
Input Int:
:: [DBG]:M:1383(0 errorhandling_ex1.ino:37)|78 0
:: [ERR](50 errorhandling_ex1.ino:25)|(50:TypeErr)120
:: [DBG]:M:1383(50 errorhandling_ex1.ino:39)|outint:-1
:: [ERR](50 errorhandling_ex1.ino:40)|(254:NoNewErr)
:: [ERR](50 errorhandling_ex1.ino:49)|(254:NoNewErr)
:: [DBG]:M:1444(50 errorhandling_ex1.ino:54)|Caught Error
:: [ERR](253 errorhandling_ex1.ino:55)|(253:ClredErr)
```

Preprossessor Definitions

You can define the following definitions to affect the behavior of this module

#define DEBUG

- define DEBUG or LOGLEVEL before importing this module to print things, otherwise error checking, etc will be silent.
- if DEBUG is defined and LOGLEVEL is not, LOGLEVEL == 50

#define LOGLEVEL

- LOGLEVEL can be defined as various values up to 50.
    - >= 50 — all messages printed (debug, info, error)
    - < 50 — debug not printed (just info and error)
    - < 40 — info not printed (just error)
    - < 30 — nothing is printed
- loglevel can also be changed while code is running with **set_loglevel(uint8_t)**. However, definning LOGLEVEL takes less programming space.

Alternate Serial Ports:

You can use a different serial port from the hardware one, through the SoftwareSerial

library. Simple initilize your SoftwareSerial class and pass it to EH_config.

Example:

```
mysoft = SoftwareSerial(10, 11); RX, TX
void setup(){
  mysoft.begin(57600); //57600 is the recommended speed because the library can delay other code
  EH_config(mysoft);
  //... rest of your code
}
```

Global Variables:

errno :: specifies error type. String format is automatically printed with log_err()
derr :: main error. This is what specifies that an error occured (non-zero == error)

Macro Overview:

Logging functions:

- **debug(...) ::**
    - prints out: [DBG](derr file:line)|...
- **log_info(...) ::**
    - prints out: [INFO](derr file:line)|...
- **log_err(...) ::**
    - prints out [ERR](derr file:line)|(strerrno:errno)|...
    - Note automatically called by all below functions except noerr and clrerr\

Error handling Functions:

Definitions:

A — assertion, if A is false the macro triggers
E -- Specified Error to be raised. See section on Defined Errors
M -- User message
R -- Return value. Should be left off for void functions.
"..." -- Same as M except can take advantage of the same functionality as Serial.println(...)

- **assert(A) ::**
    - Assert that the value is true. If it is not true, then derr and errno = ERR_ASSERT
    - Also logs error with message "AS"
    - Note: requires "error:" defined for goto
- **raise(E, ...) ::**
    - Raises the error **E** and logs it
    - Also can print a message
    - Note: requires "error:" defined for goto
- **assert_raise(A, E, ...) ::**
    - if A is false, raise(E)
    - Also can print a message

    - Note: requires "error:" defined for goto
- **iferr_catch() ::**
    - if there is an error, "goto error"
    - Note: requires "error:" defined for goto
- **iferr_log_catch() ::**
    - same as above, but also logs error.OM
    - Note: requires "error:" defined for goto
- **[fun]_return(..., return_val) ::**
    - The above functions have a "return" variety, where they return the return_val instead of "goto error"
    - They are:
        - **assert_return(A, R)** — note: in void functions, R should not be included (i.e. assert_return(A))
        - **raise_return(E, R)**
        - **assert_raise_return(A, E, R)**
        - **raisem_return(E, M, R)** — where M is a message
        - **assert_raisem_return(A, E, M, R)**
        - **iferr_return(R) iferr_log_return(R)**
- **clrerr() ::**
    - clears error specifiers (derr = 0; errno = 0
- **clrerr_log() ::**
    - clears error specifiers and logs that it did so
- **TRY(stuff) ::**
    - silences error printing if you are expecting an error.
- **CATCH_ALL ::**
    - intended to be used after try. This will catch and clear all errors. For example:

```
TRY(do_error()); // No errors will be printed
CATCH_ALL{
  // Will only happen if there was an error.
  // Note that all errors are cleared.
}
```

- **CATCH(E) ::**
    - Same as above, except it will catch a specific error E
- **CATCH_N(E) ::**
    - Will catch all errors except E
- **ELSE_CATCH(E), ELSE_CATCH_N(E) ::**
    - Same as the above two, but used after another catch (kind of like an “if else”)
- **else ::**
    - Can use at the end of a TRY, CATCH chain. For example:

```
TRY(something());  // No errors will be printed
CATCH(ERR_TYPE){
        // if there is a type error, this code will be executed
}
ELSE_CATCH(ERR_VALUE){
        // if there is a value error, this code will be executed
}
ELSE_CATCH_ALL{
        // if there was any other error (other than ERR_TYPE, ERR_VALUE), this code will be executed
}
else {
        // if there was no error, this code will be executed
}
```

Defined Errors

The following errors are defined by this module and will print the indicated output when they are raised.

Note:

ERR_NOERR indicates that no error has happened. This is the **errno** value after clrerr() is called
ERR_NONEW indicates that there are no *new* errors. **errprint** is set to this value after log_err() is called (while errno is not changed). This exists to aid in debugging so you can follow the stack of errors with each call to **iferr_log_return()**
ERR_CLEARED is a special error used by clrerr_log() and is used to let the user know where an error was cleared.

Defined	Value
ERR_NOERR	0
ERR_TIMEOUT	’ # ’
ERR_COM	’ < ’
ERR_INPUT	’ > ’
ERR_TYPE	’ T ’
ERR_VALUE	’ V ’
ERR_ASSERT	’ A ’
ERR_INDEX	’ i ’
ERR_MEMORY	’ M ’
ERR_CRITICAL	’ ! ’
ERR_CLEARED	’ - ’
ERR_NONEW	’ * ’
ERR_UNKNOWN	’ ? ’

Logging Values

The logging value can be set with either:

- #define LOGLEVEL myloglevel
- set_loglevel(myloglevel)

The *#define* version is preferred when it can be used, as it can significantly reduce your code size.

LOGV_DEBUG	**50**	**Everything is logged**
LOGV_INFO	**40**	**info and errors are logged**
LOGV_ERROR	**30**	**only errors are logged**
LOGV_SILENT	**0**	**nothing is logged**
