BCI 2000 Error Handling

Jürgen Mellinger, Gerwin Schalk

June 23, 2005

Contents

1 Handling Errors

1.1 Types of Errors

We assume that all errors we need to consider fall into one the following categories, each of which implies a different type of approach to error avoidance/error handling:

- Parameter Setup Errors
- Runtime Errors
- Logic (Programming) Errors

1.2 Parameter Setup Errors

1.2.1 Definition of the Term

This category covers anything a user can do wrong by using a program with parameters that are out of range, inconsistent, or otherwise erroneous (by, e.g., specifying an output file at a location where the user has no write permission).

Parameter setup errors, when unhandled, become runtime errors.

1.2.2 Strategies

As a guideline for approaching Parameter Setup Errors we adopt the following principle: "Whatever a user does from within an application program should never make that application crash."

In BCI 2000, this translates into a thorough parameter check done by each module before any parameter settings are actually applied to the system.

Parameter checking should comprise

- Range and Consistency checks, whereby generally ranges depend on other parameters' values;
- Signal property checks: Does the output signal of one filter meet the next filter's requirements for its input signal?
- Resource availability checks:
  - Are needed system resources available? (E.g., is it possible to open a required sound output device?)
  - Are auxiliary files (e.g., media files) available and readable?

  - Do output files have legal file names? Are output files writeable? (We could even check whether there is enough space left to write the EEG file, but practically this would not make too much sense because a concurrent process might use up the space while our system runs.)

In each of those cases, the user should get appropriate feedback guiding her towards fixing the problem.

Whenever the system tries to fix a parameter setup error by using some default set of parameters, it should do so only if

- it presents the user with a warning that tells her what it did and why it did so, and if

- the automatically fixed parameters are treated as if changed by the user, i.e. with a parameter check performed on them.

Otherwise, people might end up using a system that doesn't do what they want it to – but without telling them, so they don't ever realize –, or with a system creating new parameter inconsistencies when trying to fix others.

1.2.3 User Interface Details

The user interface for Parameter Setup Error handling is, along with the parameter setup dialog, part of the operator module. A first implementation of a GUI based user interface consists in a floating, non-modal error window popping up from the operator module that presents a list of error related textual messages to the user, allowing for browsing error messages while changing respective parameters via the parameter setup dialog. After the next parameter check, the operator module will close that window or replace its contents based on the result of the check. Parameter checking occurs when the user clicks the "SetConfig" button in the operator main window, followed by actually applying parameters in case the check was successful.

1.3 Runtime Errors

1.3.1 Definition of the Term

This category covers everything that can go wrong in the course of an application program running insofar as that malfunction is due to a lack of resources in the underlying system required for proper operation (i.e., not due to a programming error). Assuming that parameter checking has been implemented properly as outlined above, we can narrow the term 'Runtime

Error' to cases for which the following statement holds: A runtime error occurs whenever the system runs out of resources that were still available during parameter checking.

Typical reasons for this kind of error are

- the system running out of disk space while recording data,
- files being moved, trashed, or locked by a concurrent process,
- a network connection becoming unavailable.

Runtime errors, when unhandled, become logic errors because the code implies assumptions that no longer hold once a runtime error has occurred.

1.3.2 Strategies

In a properly designed and implemented system, runtime errors in the restricted sense described above will not occur frequently. However, as they are caused by undesired circumstances outside the scope of the application program itself, it seems important to provide information to the user as detailed as possible in order to enable her to prevent this type of situation in the future, and to make her aware of the fact that the application program depends on her willingness to provide a smooth operating environment. This being ensured, it seems appropriate to simply abort execution altogether, while trying to avoid a loss of the data acquired up to that time.

1.3.3 User Interface Details

In general, it is desirable to have runtime errors displayed along with the operator module's user interface. However, as this requires a working connection between the module where the error occurs, and the operator module, this may not always be possible. Therefore, in addition to an operator-based error reporting interface, each module should have a less demanding mechanism to provide error information to the user, e.g., a local log file.

1.4 Logic Errors

1.4.1 Definition of the Term

Logic, or programming, errors in general are faults of a programmer who, in his or her code, implicitly or explicitly makes assumptions that do not always hold.

1.4.2 Strategies

Programming errors are not supposed to occur at all in a tested version of an application. Therefore, instead of trying to 'handle' them, it is important to make them show up as close to their point of origin in the code as possible, by frequently and explicitly checking whether implicit assumptions actually hold, and aborting execution with an error message if this is not the case.

Aside from that, writing code as explicit, general, and simple as possible greatly reduces the possibility of making logic errors in the first place.

1.4.3 User Interface Details

As programming errors are nothing a user can do anything about, and as their occurrence with a user is some sort of glitch anyway, simply aborting the program or module with an error message seems appropriate.

2 Implementation Details

2.1 Interface to the Programmer

2.1.1 Reporting Errors

For a simple and general way to provide user communication and error reporting means to a module's programmer, there exist two global objects derived from `std::ostream`, named `bciout` and `bcierr`, in analogy to `std::cout` and `std::cerr`, where `bciout` is used to transfer general messages and warnings while `bcierr` takes actual errors.

A code example then looks like this:

```
using namespace std;
...
ofstream outputStream( fileName );
if( !outputStream.is_open() )
{
  bcierr << "Cannot open the file \""
         << fileName
         << "\" for output"
         << endl;
}
```

Furthermore, for handling runtime errors difficult to recover from, a programmer may also throw an exception that will abort execution and eventually lead to an error message being sent to the operator module (for framework related details see section 2.2):

```
...
if( ernie.find( bert ) != ernie.end() )
  throw "Ernie just ate Bert. "
        "I don't know how to tell the story.";
tellMyStory( bert.begin(), ernie.end() );
...
```

2.1.2 Checking Parameters

Checking parameters is done in a separate member function of the filter base class which, similar to the member function that does the actual processing, takes input and output signal representatives as parameters, thus allowing for signal property checking.

For the actual implementation, its declaration is as follows:

```
void GenericFilter::Preflight(
   const SignalProperties& inSignalProperties,
         SignalProperties& outSignalProperties ) const;
```

For a filter class derived from `GenericFilter`, this function is supposed to perform parameter checking as described in section 1.2.2. Instead of returning an error value, it writes possible error messages into `bcierr`. Furthermore, it communicates dimensions of its output signal which it guarantees not to exceed, and it does so by adjusting the properties of the second SignalProperties object in its argument list, e.g.

```
outSignalProperties
  = SignalProperties( inSignalProperties.Channels(), 1 );
```

or

```
outSignalProperties = SignalProperties( 0, 0 );
```

if it declares not to use its output signal.

The `const` declaration for its `this` pointer prohibits initialization functionality from `GenericFilter::Initialize()` entering into `Preflight()`; this is unwanted because it would corrupt the idea of performing a *complete* parameter check before actually *altering* the state of a filter object.

A necessary condition for a correct implementation of the `Preflight()` function is that any parameter, as well as any state that will be accessed during the processing phase, be accessed from `Preflight()` at least once. For parameters and states defined by the filter itself (i.e. inside its constructor), range and accessibility checks are automatically performed by the framework; parameters and states defined by other filters must be explicitly accessed from `Preflight()`. If a `GenericFilter` descendant fails to access an externally defined parameter or state during `Preflight()`, the first access during the processing phase will result in a runtime error.

2.1.3 Accessing Environment Objects

Parameters and states are considered to constitute an "environment", and a `GenericFilter` descendant to live in that environment, in analogy to the concept of environment variables found in some operating systems. Internally, access to the environment is mediated through a mix-in-class named `Environment` that provides accessor symbols to a filter programmer.

Low Level Access to Environment Objects is provided by the following symbols:

- `Parameters` syntactically behaves like a `PARAMLIST*`,
- `States` behaves like a `STATELIST*`,
- and `Statevector` behaves like a `STATEVECTOR*`.

As an example, take

```
float  myParameterValue = 0.0;
PARAM* param = Parameters->GetParamPtr( "MyParameter" );
if( param )
  myParameterValue = atof( param->GetValue() );
else
  bcierr << "Could not access \"MyParameter\"" << endl;
```

Unlike true pointers, these symbols cannot be assigned any values, cannot be assigned to variables, nor other manipulating operators applied. E.g., the lines

```
delete Parameters;
Parameters = new PARAMLIST;
PARAMLIST* someParamlistPointer = Parameters;
```

will all result in compiler errors.[1]

Convenient Access to Environment Objects is possible through a number of symbols which offer built-in checking and error reporting:

- `Parameter(Name[, Index 1[, Index 2]])`

  This symbol stands for the value of the named parameter. Indices may be given in numerical or textual form; if omitted, they default to 0. The type of the symbol `Parameter()` may be numerical or a string type, depending on its use.[2] If a parameter with the given name does not exist, an error message is written into `bcierr`. If the specified indices do not exist, no error is reported. In both cases, on read access, the string constant `"0"` resp. the number 0 is returned.

  Examples:

[1] In the current (preliminary) implementation, assignments from these symbols as in the last example are allowed to ease the transition process.

[2] If the compiler complains about ambiguities, use explicit typecasts as in the second example.

```
int myValue = Parameter( "MyParam" );
string myOtherValue = ( const char* )Parameter( "MyOtherParam" );
Parameter( "My3rdParam", 2, 3 ) = my3rdValue;
```

- `OptionalParameter(Default Value, Name[, Index 1[, Index 2]])`

  This symbol behaves like the symbol `Parameter()` but will not report an error if the parameter does not exist. Instead, it will return the default value given in its first argument. Assignments to this symbol are not possible.

- `State(Name)`

  This symbol allows for reading a state's value from the state vector by assigning from it, and setting a state's value in the state vector by assigning to it. Trying to access a state that is not accessible will result in an error reported via `bcierr`.

  Examples:

  ```
  short currentStateOfAffairs = State( "OfAffairs" );
  State( "OfAffairs" ) = nextStateOfAffairs;
  ```

- `OptionalState(Default Value, Name)`

  In analogy to `OptionalParameter()` this symbol does not report an error if the specified state does not exist but returns the given default value. Assignments to this symbol are not possible.

- `PreflightCondition(Condition)`

  This symbol is meant to be used inside implementations of `GenericFilter::Preflight()`. If the boolean condition given as its argument is false, it will output an error message into `bcierr` containing the condition given in its argument.

  Example:

  ```
  PreflightCondition(
    Parameter( "TransmitCh" ) <= Parameter( "SourceCh" ) );
  ```

  If TransmitCh is greater than SourceCh, a message will be sent to `bcierr` and displayed to the user, stating:[3]

  ```
  Condition not fulfilled:
    Parameter( "TransmitCh" ) <= Parameter( "SourceCh" )
  ```

[3] In future versions, the error may be reported in natural language form generated from the boolean expression.

2.2 Implementation on the Framework Side

The operator module's behaviour in response to an error message arriving from one of the modules depends on its context, i.e. on the execution phase the system is in. That way, no additional programming interface elements visible to a filter/module programmer are needed to implement an error handling scheme as described in section 1.

During the **preflight phase,** errors are Parameter Setup Errors. A module's framework code behind `bcierr` just collects error messages; on return from the preflight function, it sends those messages to the operator module which then, from the contents of the message (i.e. whether it was empty or not), determines whether the preflight was successful; on not receiving any message after some timeout[4] it assumes a broken connection or a crashed module.

During all **other phases,** the code behind `bcierr` immediately (i.e., on flushing the `std::ostream`) sends its message buffer to a log file as well as to the operator module, indicating a Runtime Error to the operator module which will, in turn, halt the system, shut down the other modules, and display the message to the user.

In addition, the top level exception handling code of each module contains similar functionality, sending an exception's associated description string into a log file and to the operator module, if possible, then quitting the module in which the exception occurred. This not only ensures a proper general handling of exceptions within the framework but also allows a programmer to handle Runtime Errors by raising her own exceptions, eliminating the need to take care of the error condition in the code following the detection of an error.

[4]For now, a simple timeout scheme with a fixed timeout interval of 5 s seems appropriate. In the future, one might consider a module requesting additional timeout periods if it expects lengthy calculations.

3 Error Handling Roadmap

- *Mellinger:* Remove references to current preliminary error handling object from code. *done Oct 21, 2002*

- *Mellinger:* Introduce `GenericFilter` inheritance into all existing filter classes. *done Mar 20, 2003*

- *Mellinger:* Create headers and dummy implementation for error stream objects in framework code to allow for using error handling from filter code; add virtual function `GenericFilter::Preflight` and a dummy implementation to allow for existing code to compile. *done Mar 21, 2003*

- *Mellinger:* Introduce calls to `GenericFilter::Preflight` into framework code. *done Apr 10, 2003*

- *Mellinger:* Implement display of error messages in operator module. *done using the status message/operator log mechanism created by* Schalk

- *Mellinger:* Write actual implementation for error stream objects that sends errors to operator. *done Apr 16, 2003*

- *Existing filters' authors:* Add implementation of `GenericFilter::Preflight` to existing filters. *done Apr 16, 2003, Mellinger*

- *Mellinger:* Remove `GenericFilter::Preflight` dummy default implementation from `GenericFilter.` *done Apr 25, 2003, Mellinger*

- *Mellinger:* Write top level exception handler that diverts exceptions occurring in modules to the operator module. *done Jun 18, 2004, Mellinger*

- *Existing filters' authors:* Fill in actual low range and high range values into parameter declarations; work out preliminary preflight implementations into complete checking.

- *Mellinger:* re-work operator and module logic to actually keep the system from running when a preflight error is reported. *done Jun 18, 2004, Mellinger*

- *Mellinger:* Enable automatic range checking for parameters; introduce a graceful system halt in case of a runtime error.