

libtrf Alarms Processor

Functional description and example

Dr. Don Goodeve, for ThinkRF

22nd January 2026

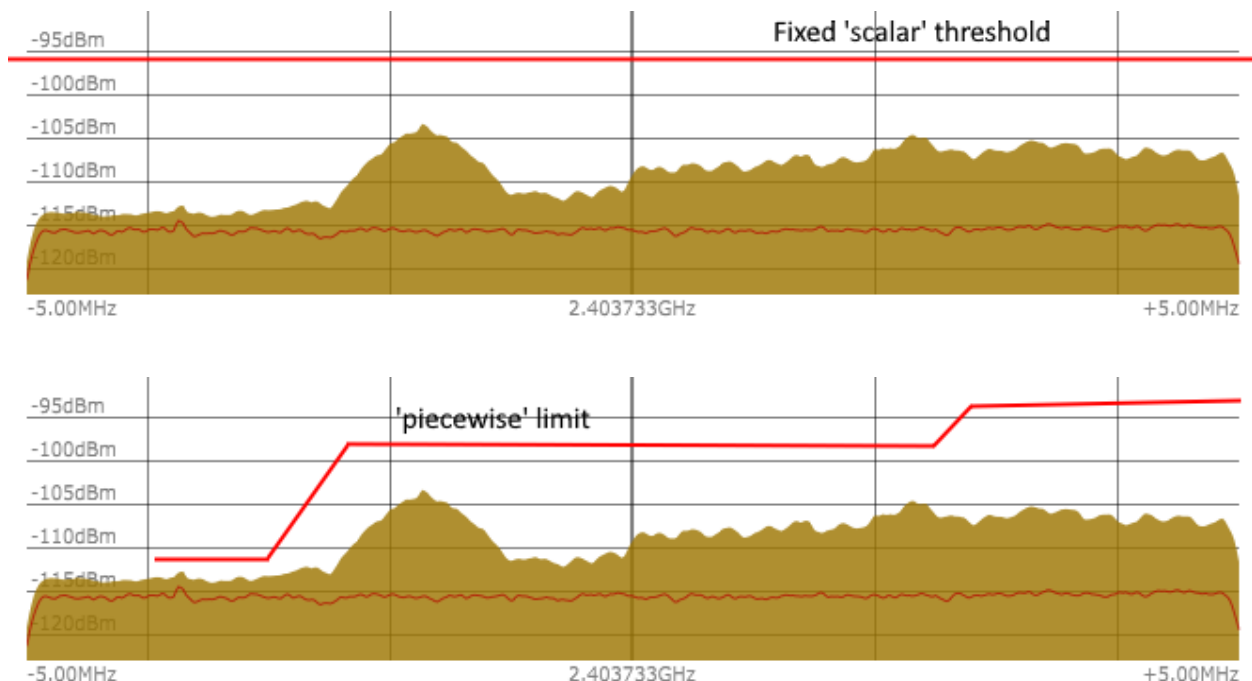
Overview

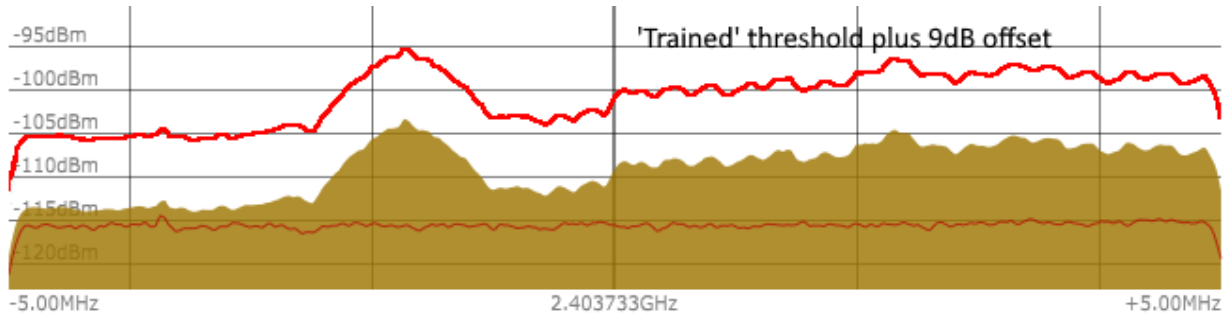
This short document outlines the 'Alarms' processor introduced into the processors set available in libtrf 1.8.1 and later. This processor provides versatile means for spectrum monitoring over multiple ranges simultaneously, providing reporting over signal 'exceedances' detected, using a number of different approaches for specifying limits; both explicitly and through 'training'.

Functional description

The 'Alarms' processor accepts a spectrum stream as input. It is programmed with one or more named ranges of interest with user-selected parameters. Once set up and supplied with a stream of spectra, the processor will continuously analyze the stream for 'exceedance' events where limits are crossed or 'loss' events when limits are no longer reached.

Limits may be set as scalar thresholds, assembled piece-wise from limit/frequency pairs, or created through time-limited training or dynamic, ongoing tracking of the signal envelope.





The incoming spectrum stream is continuously monitored to detect both signal **exceedance** and **loss** events.

Exceedance detection

If **TRFExceeddB** is non-zero for a range, it is interpreted as a margin above the specified/trained threshold that a signal must cross to be detected as an *exceedance* event.

Loss detection

If **TRFSustaindB** is non-zero for a range, it is interpreted as a margin below the specified/trained threshold that if no part of the signal within the range exceeds it, then it is registered as a signal *loss* event. Note - in the case of trained thresholds, the *lower* and *upper* limits are distinct based on the range of the observed signal over the training or tracking interval.

Reporting

Once an event (exceedance or loss) is detected, an event record is generated which may be obtained by reading back using the `trfGetParameters` interface of the `AlarmProcessor` instance. For example - JSON output for an ongoing *exceed* event:

```
{
  "exceed":[
    {
      "exceedNotLoss": true,
      "fCentreHz":680019200.000000,
      "id":0,
      "level":-38.440826,
      "name":"shaped450-500",
      "spanHz":21842.37,
      "startTimestamp":1769145084556
    }
  ]
}
```

Continued polling whilst the event persists will return a tracking update of the event. For example, the level may have increased (decreases will not be noted), the span and centre frequencies may also be refined for an ongoing event.

Once the event is *no longer detected*, a final report on the event will be provided with a populated duration parameter. For example:

```
{
  "exceed":[
    {
      "exceedNotLoss": true,
      "fCentreHz":680021200.000000,
      "id":0,
      "level":-34.2144,
      "name":"shaped450-500",
      "spanHz":24842.97,
      "startTimestamp":1769145084556
      "duration":2821
    }
  ]
}
```

Note that events are reported in an array structure, allowing multiple reports to be simultaneously provided for single or multiple programmed ranges on each polling.

Programmed thresholds

A programmed threshold is set using the TRFThreshold parameter for a range. This may contain a single value, interpreted as a scalar threshold applied across the range. Alternatively it may be specified as a *vector* of {<frequency>,<value>} pairs, describing vertices to be joined by straight lines across the range to create a threshold/signal model. If the values at extreme ends of the range are not specified, they are assumed to be identical to those of the nearest specified point.

Static trained model

A static 'trained' model may be created. This is performed by setting the TRFTrainSeconds parameter for an observation range. Once set, the processor will observe this period of spectrum and use it to create a max/min model that is then used for determining subsequent exceed and loss events. Any change of parameters for the range will reset the training.

Dynamic/adaptive model

A 'dynamic' observational mode is implemented, accessed by specifying TRFObserveSeconds for a range. In this mode, a continuously refined model (max/min) of the spectrum envelope is created with the specified length of history. Once this history is created, observation begins. Any events detected will *always* be flagged with zero duration, as the excursion giving rise to the event will itself modify the model. Any change of parameters will reset the training.

Interface description

The following describes the characteristics of the 'Alarm' processor interface:

1. The processor supports a 'spectrum stream' type input interface - ie. it may be attached to a libtrf spectrum stream source such as a receiver, or another processor such as *IQToSpectrum*.
2. The processor is controlled using the libtrf conventional 'setParameters' JSON interface. This interface supports the following programming parameters, errors in which will result in rejection of 'setParameters':
 - a. **TRFAddRange**: Specifies a new range of frequencies to be added to the set managed by the processor. A 'newRange' consists of one or more (array of) objects containing:
 - i. **TRFRangeName**: A textual name for the range assigned by the user.
 - ii. **TRFRangeActive**: (optional - default 'true') whether the range is active.
 - iii. **TRFFMinHz**: Start frequency
 - iv. **TRFFMaxHz**: Stop frequency
 - v. **TRFUnits**: (optional) specifies the units to be used for threshold calculations.
 - vi. **TRFExceeddB**: (optional) specifies the amount by which the threshold must be exceeded (in dB) for an alarm event to be generated. Default +6dB.
 - vii. **TRFSustaindB**: (optional) specifies the amount by which a signal must *consistently drop below* the threshold *across the entire programmed range* for a signal loss event to be registered. Default 0.0dB (ie. disabled). This value is specified in negative dB.
 - viii. **TRFThreshold**: (optional¹) One of the following forms of threshold descriptor:
 1. A scalar value - interpreted as an absolute level.
 2. A vector of pairs of 'frequency' and 'level' pairs specifying a linear piece-wise threshold boundary, specified in monotonically rising frequency.
 - ix. **TRFTrainSeconds**: (optional) Specifies a training time to be used to create a threshold that will be subsequently applied.
 - x. **TRFObserveSeconds**: (optional) Specifies a continuously adapting (dynamic) trained range with a specified observation period².
 - b. **TRFRemoveRange**: Specifies the name of a range to be removed from the set managed by the processor. It is an error for the range not to exist.

¹ One of TRFThreshold, TRFTrainSeconds or TRFObserveSeconds *must* be specified for TRFAddRange' to be accepted by 'setParameters'. Note that each 'setParameters' call is accepted or rejected atomically.

² Whilst a very versatile tool, this is potentially memory and cpu-hungry. Also, because of the dynamic update characteristic, it will report the start of an exceedance and then will adapt to the change in the spectrum - thus all events reported will be 'zero' duration.

- c. **TRFModifyRange:** Allows parameters of the named range to be modified. It is an error for the range not to exist.
 - i. If modifyRange 'touches' training parameters, then this will force a retraining to occur.
 - ii. If modifyRange 'touches' observation time, then the current observation will be reset.
3. The processor is observed using the libtrf conventional 'getParameters' interface. If there are active exceed records, this will report them in a JSON array named TRFExceedRecords containing the following parameters:
 - a. **TRFEventID:** A unique ID for the specific event.
 - b. **TRFExceedNotLoss:** A boolean flag indicating the type of event.
 - c. **TRFRangeName:** The user-supplied name of the range in which the exceedance was observed.
 - d. **TRFFCentreHz:** The centre frequency of the observed excursion.
 - e. **TRFSpan:** The bandwidth of the excursion (all contiguous frequencies violating the limit).
 - f. **TRFLevel:** In the current units, the absolute level of the maximum exceeding excursion (positive or negative relative to the limit).
 - g. **TRFStartTimeStamp:** The timestamp of the event onset in milliseconds.
 - h. **TRFEventDuration:** For events that have 'finished', the observed duration. If this parameter is not present, the event is ongoing.

Examples:

The following specifies a region from 250-350MHz with a fixed positive detection threshold of -40dBm.

```
// Alarm region 250-350MHz, fixed positive threshold (-40dBm)
char *pProgram(nullptr), *pRange(nullptr);
trfAddTextParameter(&pRange, TRFRangeName, "fixed250-350");
trfAddNumericParameter(&pRange, TRFFMinHz, 250.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 350.0e6f);
trfAddNumericParameter(&pRange, TRFThreshold, -40.0f);
trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." << endl;
    return false;
}
```

The following specifies a region from 450-500MHz with a shaped model and asymmetric exceed and sustain margins.

```
// Alarm region 450-500MHz, shaped thresholds - positive and negative
trfAddTextParameter(&pRange, TRFRangeName, "shaped450-500");
trfAddNumericParameter(&pRange, TRFFMinHz, 450.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 500.0e6f);
trfAddNumericParameter(&pRange, TRFExceeddB, +3.0f); // Exceed margin
trfAddNumericParameter(&pRange, TRFSustaindB, -20.0f); // Sustain margin
```

```

std::list<_Limit> lModel({
    {460.0e6f, -50.0f}, {470.0e6f, -35.0f}, {480.0e6f, -35.0f}, {490.0e6f, -50.0f}
});
char *pElement(nullptr);
for (const auto &cRange : lModel) {
    trfAddNumericParameter(&pElement, TRFFrequency, cRange.fFrequency);
    trfAddNumericParameter(&pElement, TRFLevel, cRange.fLevel);
    trfAddJSONElementToArray(&pRange, TRFThreshold, pElement);
    trfDisposeString(&pElement);
}
trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." << endl;
    return false;
}

```

The following specifies a ‘trained’ threshold region, constructed from the first 10 seconds of spectra delivered, from 550-620MHz with a +10dB triggering margin:

```

// Add trained threshold - 10 seconds training 550-620MHz with +12dB triggering margin
trfAddTextParameter(&pRange, TRFRangeName, "trained550-620");
trfAddNumericParameter(&pRange, TRFFMinHz, 550.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 620.0e6f);
trfAddNumericParameter(&pRange, TRFTrainSeconds, 10.0f);
trfAddNumericParameter(&pRange, TRFExceeddB, 12.0f); // Applies +ve
trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." << endl;
    return false;
}

```

The following specifies a dynamic ‘observing’ detector region from 680-740MHz with a +12dB triggering margin and a 30-second observation history:

```

// Add dynamic tracking threshold 680-740MHz - 30 seconds history tracked
// with +12dB triggering margin
trfAddTextParameter(&pRange, TRFRangeName, "dynamic680-740");
trfAddNumericParameter(&pRange, TRFFMinHz, 680.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 740.0e6f);
trfAddNumericParameter(&pRange, TRFObserveSeconds, 30.0f);
trfAddNumericParameter(&pRange, TRFExceeddB, 12.0f); // Size of +ve excursion required
trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." << endl;
    return false;
}

```

Complete example

The following code is a complete stand-alone compilable example, included as AlarmsProcessorExample.cpp in the examples release directory. The code configures the Alarms processor, sets up a spectrum stream and starts observation, reporting results every second. The output may be interrupted by the user.

```
/**
 * @example AlarmsProcessorExample.cpp
 *
 * Example code demonstrating the operation of the Alarms processor
 * 1. Set up Alarms processor with a set of regions using different alarm region designs.
 * 2. Report spectral events in each of the regions in real-time.
 * 3. Terminate on user Ctrl-C
 *
 * Language: c++17
 * Code Copyright (C) 2025 ThinkRF Incorporated. All Rights Reserved.
 */

#include <iostream>
#include <string>
#include <list>
#include <thread>
#include <chrono>

// Cross platform standard utilities
#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif
#include <filesystem>

#ifdef __linux
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#endif

#include "libtrf.h"

using namespace std;

// Useful local functions and classes
//*****
#ifdef _SIGNINTHANDLER
// Signal handling logic for user controlled asynchronous (Ctrl-C) application abort
std::atomic<bool> gbShutdownRequested(false);
static void _handleSIGINT(int iSignal) {
    if (SIGINT == iSignal) {
        gbShutdownRequested.store(true);
    }
}

static void _installSignalHandler(void) {
    std::signal(SIGINT, _handleSIGINT);
}
#define _SIGNINTHANDLER (1)
#endif

#ifdef _CLEANUPHANDLER
```

```

// RAII exit cleanup pattern
class _Cleanup {
private:
    std::list<std::function<void(void)>> mlCleanup;

public:
    void add(std::function<void(void)> cFn) { mlCleanup.push_front(cFn); }

    ~_Cleanup() {
        for (auto cFn : mlCleanup) {
            if (cFn != nullptr) {
                cFn();
            }
        }
        mlCleanup.clear();
    }
};

#define _CLEANUPHANDLER
#endif

// Main application function
// The following code performs the scan/capture sequence:
// 0. Setup libtrf for capture. Establish streams and processors, opens
//    receiver etc.
// 1. Sets up four 'alarm' regions within stream 200-800MHz capture range
//    a. 250-350MHz - fixed threshold at -40dBm
//    b. 450-500MHz - shaped threshold up to -40dBm.
//    i. 'Sustain' threshold specified (close to noise floor)
//    c. 550-620MHz - 'trained' threshold (10 seconds training)
//    d. 680-740MHz - 'dynamic' threshold with 30 seconds observation
// 2. Starts capture
//    a. Every second, reports any events
//    d. Capture continues until interrupted by user (Ctrl-C) - SIGABRT
//*****
bool AlarmProcessorExample(int argc, const char *argv[]) {
    cout << "Opening device..." << endl;        // Open 0th device detected

    trfAddDeviceAddress("192.168.10.100");        // Example of explicit add for device

    _uint uDevices(0);
    trfStatus eStatus(trfEnumerateDevices(&uDevices, true));
    if ((eStatus != trfOk) || (uDevices == 0)) {
        cout << "Did not find any ThinkRF devices. Exiting." << endl;
        return false;
    }

    // Cleanup - executes on scope exit
    _Cleanup cCleanup;

    // Useful structures
    struct _Limit {
        float fFrequency;
        float fLevel;
    };

    // Open device
    trfHandle hDevice(trfInvalidHandle);
    eStatus = trfOpenDevice(0, &hDevice);
    if (eStatus != trfOk) {
        cout << "Open device failed with code: " << eStatus << ". Exiting." << endl;
        return false;
    }
    cCleanup.add([&hDevice]() {trfClose(&hDevice); }); // Add cleanup function

```

```

// Build processing infrastructure. IQStream, IQToSpectrumProcessor
cout << "Creating sampling infrastructure..." << endl;
trfHandle hSpectrumStream(trfInvalidHandle);
eStatus = trfCreateSpectrumStream(
    &hSpectrumStream, 200.0e6, 800.0e6, 10.0e3f, "nuttall", -30.0f);
if (eStatus != trfOk) {
    cout << "trfSpectrumStream failed with code: " << eStatus << ". Exiting." <<
endl;
    return false;
}
cCleanup.add([&hSpectrumStream]() {trfClose(&hSpectrumStream); });

trfHandle hAlarmProcessor(trfInvalidHandle);
eStatus = trfCreateProcessor(&hAlarmProcessor, "Alarms");
if (eStatus != trfOk) {
    cout << "trfCreateProcessor failed with code: " << eStatus <<
        ". Exiting." << endl;
    return false;
}
cCleanup.add([&hAlarmProcessor]() {trfClose(&hAlarmProcessor); });

// Alarm region 250-350MHz, fixed positive threshold (-40dBm)
char *pProgram(nullptr), *pRange(nullptr);
trfAddTextParameter(&pRange, TRFRangeName, "fixed250-350");
trfAddNumericParameter(&pRange, TRFFMinHz, 250.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 350.0e6f);
trfAddNumericParameter(&pRange, TRFThreshold, -40.0f);
trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." <<
endl;
    return false;
}

// Alarm region 450-500MHz, shaped thresholds - positive and negative
trfAddTextParameter(&pRange, TRFRangeName, "shaped450-500");
trfAddNumericParameter(&pRange, TRFFMinHz, 450.0e6f);
trfAddNumericParameter(&pRange, TRFFMaxHz, 500.0e6f);
trfAddNumericParameter(&pRange, TRFExceeddB, +3.0f); // Exceed margin
trfAddNumericParameter(&pRange, TRFSustaindB, -20.0f); // Sustain margin

std::list<_Limit> lModel({
    {460.0e6f, -50.0f}, {470.0e6f, -35.0f}, {480.0e6f, -35.0f}, {490.0e6f, -50.0f}
});
char *pElement(nullptr);
for (const auto &cRange : lModel) {
    trfAddNumericParameter(&pElement, TRFFrequency, cRange.fFrequency);
    trfAddNumericParameter(&pElement, TRFLevel, cRange.fLevel);
    trfAddJSONElementToArray(&pRange, TRFThreshold, pElement);
    trfDisposeString(&pElement);
}

trfAddJSONElement(&pProgram, TRFAddRange, pRange);
trfDisposeString(&pRange);
eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." <<
endl;
    return false;
}

// Add trained threshold - 10 seconds training 550-620MHz with +12dB triggering margin
trfAddTextParameter(&pRange, TRFRangeName, "trained550-620");

```

```

    trfAddNumericParameter(&pRange, TRFFMinHz, 550.0e6f);
    trfAddNumericParameter(&pRange, TRFFMaxHz, 620.0e6f);
    trfAddNumericParameter(&pRange, TRFTrainSeconds, 10.0f);
    trfAddNumericParameter(&pRange, TRFExceeddB, 12.0f);          // Applies +ve
    trfAddJSONElement(&pProgram, TRFAddRange, pRange);
    trfDisposeString(&pRange);
    eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
    if (eStatus != trfOk) {
        cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." <<
endl;
        return false;
    }

    // Add dynamic tracking threshold 680-740MHz - 30 seconds history tracked
    // with +12dB triggering margin
    trfAddTextParameter(&pRange, TRFRangeName, "dynamic680-740");
    trfAddNumericParameter(&pRange, TRFFMinHz, 680.0e6f);
    trfAddNumericParameter(&pRange, TRFFMaxHz, 740.0e6f);
    trfAddNumericParameter(&pRange, TRFObserveSeconds, 30.0f);
    trfAddNumericParameter(&pRange, TRFExceeddB, 12.0f);          // +ve excursion required to
trigger
    trfAddJSONElement(&pProgram, TRFAddRange, pRange);
    trfDisposeString(&pRange);
    eStatus = trfSetParameters(hAlarmProcessor, &pProgram);
    if (eStatus != trfOk) {
        cout << "trfSetParameters failed with code: " << eStatus << ". Exiting." <<
endl;
        return false;
    }

    // Select which we run with. Comment out the one(s) you want to test
    std::list<std::string> lRemove({
        //"fixed250-350",
        //"shaped450-500",
        //"trained550-620",
        //"dynamic680-740",
    });
    for (auto &sName : lRemove) {
        trfAddTextParameter(&pProgram, TRFRemoveRange, sName.c_str());
        trfSetParameters(hAlarmProcessor, &pProgram);
    }

    // Processor now programmed - connect processor to stream, and stream to receiver
    eStatus = trfAttachProcessorToStream(hAlarmProcessor, hSpectrumStream);
    if (eStatus != trfOk) {
        cout << "trfAttachProcessorToStream failed with code: " << eStatus
            << ". Exiting." << endl;
        return false;
    }

    // Start processing (Rx -> alarm processor)
    eStatus = trfAttachStreamToDevice(hDevice, hSpectrumStream);
    if (eStatus != trfOk) {
        cout << "trfAttachProcessorToStream failed with code: " << eStatus <<
            ". Exiting." << endl;
        return false;
    }

    // Results monitoring loop
    _installSignalHandler();          // SIGABRT handler
    cout << "Scanning loop starts (Ctrl-C to exit)" << endl;
    char *pResults(nullptr);
    while (false == gbShutdownRequested) {
        std::this_thread::sleep_for(std::chrono::seconds(1));

```

```
        eStatus = trfGetParameters(hAlarmProcessor, &pResults);
        if (pResults != nullptr) {
            cout << pResults << endl;
            trfDisposeString(&pResults);
        }
        else {
            cout << "No results" << endl;
        }
    }

    cout << "User interrupt. Shutting down." << endl;

    return true;
}

// Main application entry-point
//*****
#ifdef _MAIN_
#define _MAIN_
int main(int argc, const char *argv[]) {
    cout << "libtrf example code " << __FILE__ << endl << "---" << endl;
    cout << "library version " << trfGetVersion() << endl << endl;

    AlarmProcessorExample(argc, argv);

    cout << "Example complete" << endl;
}
#endif // _MAIN_
```