ERICSSON

Trace Code Instrumentation Instructions

PROGRAMMER'S GUIDE

1/198 17-APR 901 0500/1 Uen PA15

Contents

1 About This Guide

This document describes how to code Component Based Architecture (CBA) applications for tracing using the Trace Common Component (CC).

1.1 Intended Audience

This guide is intended for developers of CBA common components and applications.

1.1.1 Prerequisite Knowledge

Users of this document should have knowledge and experience of the following:

- C/C++ programming principals
- Trace *Key Concepts* and naming guidelines documented in *Trace Event Guidelines* (Reference [1])
- Linux
- Automake

1.2 How This Guide is Organized

This document is organized into the following major sections:

Table 1 Document Organization

Section	Description
About This Guide	Introduces the guide, describing prerequisites, document structure, and the conventions used.
Overview	Provides an overview of code instrumentation and describes the activities that are involved in the instrumentation process.
Working with Tracepoints	Describes how to add tracepoint statements to the program code and how to create a tracepoint provider header file.
Preparing the Tracepoint Probe	Describes the creation of a probe function for the instrumented application.
Compiling Instrumented Application	Describes how to compile instrumented applications.

1.3 Conventions Used in This Guide

Table 2 provides a list of typographic conventions that may be encountered in this document:

Table 2 Typographic Conventions

Convention	Description	Example
Code Examples	Code examples	`stat char* months[] ={"Jan","Feb"}`
Command Variables	Command variables, the values of which you must supply	***<home_directory>***
Document and File Names	References to document titles or sections in a document and file names	For more information, refer to the *System Administrator Guide*. Check the local runlog files (*xxx.runlog* and *xxa.runlog*) in the `/var/log/xxx` directory.
GUI Objects	GUI objects, such as menus, fields, and buttons, dialog boxes, and options	On the **File** menu, click **Exit**.
Key Combinations	Key combinations	Press **Ctrl+X** to delete the selected value.(1)
Output Information	Text displayed by the system	`System awaiting input`
Parameter/Configuration Values	Parameter values (numbers, true/false, yes/no, and so on)	To use this feature, the parameter must be set to *true*.
System Elements	Command and parameter names, program names, path names, URLs, and directory names	The files are located in `E:\Test`. The files are located in `etc/opt/ericsson/bin`.(2)

Convention	Description	Example
User Input	A command that you must enter in a Command Line Interface (CLI) exactly as written	`cd $HOME`
Line Break	The arrow symbol (⇒) can be used when an inappropriate line break has been made. An inappropriate line break occurs when the code lines are too long to fit on the page, and there is no appropriate place for a line break.	`>cd /opt/msmw-cds-⇒ cxp-<version>`

(1) The plus sign (+) indicates that you must press the keys simultaneously.
(2) The use of the forward slash (/) is for UNIX systems; PC systems use the backslash (\).

1.4 Prerequisites

Trace Environment Adapter (EA) Software Development Kit (SDK) has been installed on the development machine.

For installation procedures, refer to the *Trace EA Installation Instructions* (Reference [2])

1.5 Comments About the Documentation

Ericsson encourages you to provide feedback, comments or suggestions so that we can improve the documentation to better meet your needs. With your comments provide the following:

- Document title
- Document number and revision
- Page number

Please send your comments to your local Ericsson Support.

2 Overview

The Trace Common Component (CC) is a User-Space Tracer (UST) that collects and processes trace information from applications at runtime. Trace information is generated by tracepoint statements that must be added to the program code. The process of coding an application for tracing is called instrumenting the application. This document describes the instrumentation process for C/C++ based applications.

Instrumenting your application involves the following activities:

1. Adding tracepoint statements to the program code, see Section 3.2 on page 14.
2. Creating one or more tracepoint provider header files that define the tracepoint statements, see Section 3.1 on page 5.
3. Preparing the tracepoint probe, see Section 4 on page 17.
4. Modifying the *Makefile.am* to compile the instrumented application, see Section 5 on page 18.

Trace CC uses the Linux Trace Toolkit Next Generation (LTTng) UST to provide the trace infrastructure. Instrumenting an application to use Trace CC functionality requires the inclusion of several dependant LTTng files. These include files are installed with the Trace Environment Adapter (EA) Software Development Kit (SDK). The Trace EA SDK must be installed on the development machine to instrument your application. For more information on installing the Trace EA SDK, refer to the *Trace EA Installation Instructions*, (Reference [2]).

Once compiled, the instrumented application requires LTTng libraries to offer trace functionality. These libraries are installed in the target machine with Trace EA.

3 Working with Tracepoints

Tracepoints resemble function calls that are added to the application source code for tracing purposes. When an active tracepoint is hit during the execution of an instrumented application, arguments specified in the corresponding tracepoint declaration are captured and stored in the output files. Tracepoints provide a hook to call a probe function at runtime. This hooking mechanism is

dynamic and can be activated or deactivated during a particular trace session. The hook and the probe function are handled by the shared UST library and will remain transparent during code instrumentation.

When active, the probe takes arguments passed by the associated tracepoint and writes them to an event in the trace buffers. Inactive tracepoints are processed transparently. For more information on tracepoint overhead, refer to the LTTng UST documentation.

Note: Tracepoints can be added to all types of C/C++ code.

Multi-threaded programs, signal handlers, and shared libraries are all supported.

Adding tracepoints to your code requires:

- Tracepoint declarations in one or more tracepoint provider header files.
- Tracepoint statements in your program code.

3.1 Adding Tracepoint Declarations

The first part of code instrumentation is the creation of one or more tracepoint provider header files that define the tracepoints in your program code. These tracepoint definitions are used to set up events that can be written to the trace buffers.

All tracepoints defined within a tracepoint provider header file must belong to the same domain. These domains were assigned when coding the individual tracepoint statements. For more information, refer to **tracepoint_provider** in Section 3.2 on page 14.

Because each header file defines the events for a single domain, multiple files are often necessary. To help keep track of these files, the filename for each header file should be the same as the `TRACEPOINT_PROVIDER` defined within it, including a *.h* extension.

For example:

If `com_ericsson_aplicationx` is the tracepoint provider for the events in a header file, then the corresponding filename should be *com_ericsson_applicationx.h*.

Example 1 presents a sample tracepoint provider header file.

```
#undef TRACEPOINT_PROVIDER
#define TRACEPOINT_PROVIDER com_ericsson_applicationx


#undef TRACEPOINT_INCLUDE_FILE
#define TRACEPOINT_INCLUDE_FILE ./com_ericsson_applicationx.h


#ifdef __cplusplus
extern "C" {
#endif


#if !defined(_COM_ERICSSON_APPLICATIONX_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _COM_ERICSSON_APPLICATIONX_H


#include <lttng/tracepoint.h>


TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName1,
            TP_ARGS(int, LoopCounter_i,
                    int, MyInt
                    ),
            TP_FIELDS(
                      ctf_integer(int, MyLoopCounter_i, LoopCounter_i)
                      ctf_integer(int, MyInt, MyInt)
                     )
)


TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName1, TRACE_WARNING)


TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName2,
            TP_ARGS( const char *, MyString
                    ),
            TP_FIELDS(ctf_string(string, MyString)
                     )
)


TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName2, TRACE_WARNING)


TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName3,
            TP_ARGS( char *, array_text,
                     size_t, array_text_length,
                     int, MyNewLength,
                     int, MyOverflowLength
                     ),
            TP_FIELDS(ctf_array(char, array, array_text, 3)
                      ctf_array(char, array2, array_text, 1)
                      ctf_array_text(char, arrayText, array_text, 2)
                      ctf_array_text(char, arrayText2, array_text, 20)
                      ctf_sequence(char, Sequence, array_text, size_t,array_text_length)
                      ctf_sequence_text(char, sequenceText, array_text, size_t,array_text_length)
                      ctf_sequence_text(char, sequenceText2, array_text, size_t,MyNewLength)
                      ctf_sequence_text(char, sequenceText3, array_text, size_t,MyOverflowLength)
                      ctf_string(String, array_text)
                     )
)
```

```
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName3, TRACE_WARNING)
```

```
#endif /* _COM_ERICSSON_APPLICATIONX_H */

/* This part must be outside protection */
#include <lttng/tracepoint-event.h>

#ifdef __cplusplus
}
#endif
```

Example 1 Sample Tracepoint Provider Header File

This example highlights the elements of a tracepoint provider header file. Each element is described in the following sections.

3.1.1 TRACEPOINT_PROVIDER

```
#undef TRACEPOINT_PROVIDER
#define TRACEPOINT_PROVIDER com_ericsson_applicationx
```

The `TRACEPOINT_PROVIDER` macro specifies the tracing domain for events defined in the header file.

Because the provider is part of each tracepoint event name, it must adhere to a standardized naming convention to ensure that there are no tracepoint name collisions with other instrumented programs. For more information on naming conventions and guidelines, refer to the *Trace Event Guidelines* (Reference [1]).

3.1.2 TRACEPOINT_INCLUDE_FILE

```
#undef TRACEPOINT_INCLUDE_FILE
#define TRACEPOINT_INCLUDE_FILE ./com_ericsson_applicationx.h
```

The `TRACEPOINT_INCLUDE_FILE` is the name and location of this header file, including the file extension.

Header files should be named after the corresponding `TRACEPOINT_PROVIDER` with a *.h* extension.

Note: The path can be relative or absolute. Use '`.`' to specify the current directory.

3.1.3 extern "C"

```
#ifdef __cplusplus
#extern "C"{
#endif

...

#ifdef __cplusplus
}
#endif
```

The `TRACEPOINT_EVENT` macro defines C functions. If you are coding in C++, you must surround your header with `extern "C"` to ensure that it compiles correctly. Using a `ifdef __cplusplus` preprocessor directive allows you to implement mixed code using C and C++ functions.

3.1.4 Header File Definition

```
#if !defined(_COM_ERICSSON_APPLICATIONX_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _COM_ERICSSON_APPLICATIONX_H

..

#endif
```

To avoid double definition of `TRACEPOINT_EVENT` macros, enclose them inside this `if` statement.

Note: The UST `TRACEPOINT_HEADER_MULTI_READ` macro allows the trace infrastructure to read the header file multiple times to define all functions and data structures it needs based on a single definition.

3.1.5 tracepoint.h

```
#include <lttng/tracepoint.h>
```

`lttng/tracepoint.h` is part of the LTTng-UST framework. It must be included in your tracepoint provider header file to enable UST tracing.

3.1.6 TRACEPOINT_EVENT

```
TRACEPOINT_EVENT(sample_domain, event_name,
  TP_ARGS([type,var_name]
         [,type,array_name] // If ctf_string is used in TP_FIELDS.
         [,type,array_name ,size_t,array_length] // If ctf_sequence or ctf_sequence_text
           ...                                   // is used in TP_FIELDS.
           ...
          ),
  TP_FIELDS(
      [ctf_integer(int, label, var_name)]
      [ctf_integer_hex(int, label, var_name)]
      [ctf_integer(long, label, var_name)]
      [ctf_integer_network(int, label, var_name)]
      [ctf_integer_network_hex(int, label, var_name)]
      [ctf_array(long, label, array_name, print_length)]
      [ctf_array_text(char, label, array_name, print_length)]
      [ctf_sequence(char, label, array_name, count_type, array_length)]
      [ctf_sequence_text(char, label, array_name, count_type, array_length)]
      [ctf_string(label, array_name)]
      [ctf_float(float, label, var_name)]
      [ctf_float(double, label, var_name)]
           )
)
```

Each `tracepoint()` statement in your program code must map to a `TRACEPOINT_EVENT` definition in the tracepoint provider header file. Each `TRACEPOINT_EVENT` has three parts:

- Name declaration
- Argument listing
- Fields listing

Name Declaration

```
TRACEPOINT_EVENT(tracepoint_provider, event_name, ...)
```

The name declaration consists of two fields: the tracepoint provider and a user-defined event name. The event name must be a valid c variable name. Event names are limited to 127 characters and must begin with a letter or an underscore. Digits are supported after the initial character.

Argument Listing

```
TP_ARGS([type,var_name]
       [,type,array_name] // If ctf_string is used in TP_FIELDS.
       [,type,array_name ,size_t,array_length] // If ctf_sequence or ctf_sequence_text
         ...                                   // is used in TP_FIELDS.
),
```

The argument field (`TP_ARGS`) defines the arguments passed by the corresponding tracepoint statement. The `type` and `name` are separated by commas.

Note: When printing output from an array using `ctf_sequence` or `ctf_sequence_text`, the array length must be included as a separate value using `size_t,array_length`.

This macro must always be included with the tracepoint declaration. If the corresponding tracepoint statement does not pass any arguments, include the macro, but leave it empty as follows:

```
TP_ARGS(),
```

Fields Listing

```
TP_FIELDS(
    [ctf_integer(int, label, var_name)]
    [ctf_integer_hex(int, label, var_name)]
    [ctf_integer(long, label, var_name)]
    [ctf_integer_network(int, label, var_name)]
    [ctf_integer_network_hex(int, label, var_name)]
    [ctf_array(long, label, array_name, print_length)]
    [ctf_array_text(char, label, array_name, print_length)]
    [ctf_sequence(char, label, array_name, count_type, array_length)]
    [ctf_sequence_text(char, label, array_name, count_type, array_length)]
    [ctf_string(label, array_name)]
    [ctf_float(float, label, var_name)]
    [ctf_float(double, label, var_name)]
        )
```

The fields listing (`TP_FIELDS`) carries the values that are going to appear in the trace printout for this tracepoint.

Common Trace Format (CTF) macros are used to print data. Each CTF macro identifies a fieldname=value pair in the trace printout. Together, the CTF macros for a `TRACEPOINT_EVENT` form the payload of the tracepoint. These macros convert trace data into the binary Common Trace Format. This format is used to reduce the storage space needed for trace output.

Note: Square brackets indicate optional elements.

`TP_FIELDS` must always be included with the tracepoint declaration. If the corresponding tracepoint statement does not pass any arguments, include the macro, but leave it empty as follows:

```
TP_FIELDS(),
```

The following table lists the main CTF macros:

Table 3 CTF Macros

<CTF macro>	Description
ctf_integer	Stores a decimal integer in the digit order specified by the traced machine.

Table 3 CTF Macros

<CTF macro>	Description
ctf_integer_hex	Stores a hexadecimal integer in the digit order specified by the traced machine.
ctf_integer_network	Stores a decimal integer in network order.
ctf_integer_network_hex	Stores a hexadecimal integer in network order.
ctf_float	Stores a decimal floating point number as specified by the traced machine.
ctf_array	Stores a **fixed** size collection of values. Used to print out a fixed number of elements that are known at coding time. Because the number of elements to print is known in advance, there is no need to pass the array length when calling the tracepoint() macro. Only primitive array types are supported. These include `int`, `long`, `float`, and `char`. **Note:** If the array type is `char`, the printout of each element will be an integer representation of the character stored in that element. For example: "test" will be printed as: `[0] = 116, [1] = 101, [2] = 115, [3] = 116`.
ctf_array_text	Stores a **fixed** size collection of characters. Characters are displayed up to a null. Used to print out a fixed number of characters that are known at coding time. Because the number of elements to print is known in advance, there is no need to pass the array length when calling the tracepoint() macro. **Note:** If the array type is `char`, the printout of each element will be an integer representation of the character stored in that element. For example: "test" will be printed as: `[0] = 116, [1] = 101, [2] = 115, [3] = 116`.

Table 3 CTF Macros

<CTF macro>	Description
ctf_sequence	Stores a **variable** size collection of values. Used to print out a variable number of elements that are not known at coding time. Because the number of elements to print is not known in advance, the array length must be passed when calling the tracepoint() macro. Only primitive array types are supported. These include `int`, `long`, `float`, and `char`. **Note:** If the array type is `char`, the printout of each element will be an integer representation of the character stored in that element. For example: "test" will be printed as: `[0] = 116, [1] = 101, [2] = 115, [3] = 116`.
ctf_sequence_text	Stores a **variable** size collection of characters. Characters are displayed up to a null. Used to print out a variable number of elements that are not known at coding time. Because the number of elements to print is not known in advance, the array length must be passed when calling the tracepoint() macro. **Note:** Only arrays of type `char` are supported. Each element will be printed out in plain text. For example: "test" will be printed as: "`test`".
ctf_string	Stores a variable size collection of characters that does not require a user-specified length. Characters are stored up to a null.

The syntax for TP_FIELDS is:

```
TP_FIELDS(
        [<CTF macro>(<type>,<field name>,<value>[,<count type/array_length>,<element count>])]
```

Where:

<type> is the argument type for the incoming argument. Only primitive types, structures, and pointers are supported.

`int`, `long`, `float`, `char`, `struct`, including arrays and pointers of these types.

Note: For more information on using C structures in a tracepoint statement, refer to Section 7.1 on page 32.

<field_name> is the field name that will appear in the trace event printout. The *<field name>* is an unquoted, literal string that supports text without spaces.

Note: The field_name must be unique within the corresponding header file.

<value> is the value that will be stored for this field. The *<value>* must adhere to the same *<type>* as specified in the *<CTF macro>*. It can be a variable or a literal.

<count_type> Only used for sequences (ctf_sequence and ctf_sequence_text).

Specifies a type for the *<print length>*.

<print_length> Only used for arrays (ctf_array and ctf_array_text).

Is a fixed length, literal integer that specifies the number of elements in *<value>* to print.

<array_length> Only used for sequences.

Is a variable that specifies the number of elements in <value> to print. This variable is passed from the prototype definition.

3.1.7 TRACEPOINT_LOGLEVEL

```
...
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName1, TRACE_WARNING)
...
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName2, TRACE_WARNING)
...
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName3, TRACE_WARNING)
...
```

TRACEPOINT_LOGLEVEL has the following format:

```
TRACEPOINT_LOGLEVEL(<tracepoint provider>,<event name>,<log level name>)
```

TRACEPOINT_LOGLEVEL is an optional macro that assigns a log level to the corresponding tracepoint event. Log levels indicate the expected frequency of trace output, providing an additional way to categorize tracepoint events. Lower log levels are intended for more critical, but less frequent tracepoint events.

Note: The TRACEPOINT_LOGLEVEL macro must be defined in the same header file as the corresponding tracepoint event.

Table 4 describes the 15 predefined trace log levels that are available for use.

Table 4 Trace Log Levels

Name	Level
TRACE_EMERG	0
TRACE_ALERT	1
TRACE_CRIT	2
TRACE_ERR	3
TRACE_WARNING	4
TRACE_NOTICE	5
TRACE_INFO	6
TRACE_DEBUG_SYSTEM	7
TRACE_DEBUG_PROGRAM	8
TRACE_DEBUG_PROCESS	9
TRACE_DEBUG_MODULE	10
TRACE_DEBUG_UNIT	11
TRACE_DEBUG_FUNCTION	12
TRACE_DEBUG_LINE(1)	13
TRACE_DEBUG	14

(1) TRACE_DEBUG_LINE is the default log level assigned to any tracepoint that does not have an explicit log level assignment in the corresponding header file.

3.1.8 tracepoint-event.h

```
#include <lttng/tracepoint-event.h>
```

`lttng/tracepoint-event.h` is part of the Lttng-UST framework. It must be included in your header file to enable UST tracing. It should be specified at the end of the tracepoint provider header file.

3.2 Adding Tracepoint Statements

The second part of code instrumentation is the modification of program code for software tracing. These modification include the addition of tracepoint statements. Tracepoint statements allow you to print information, such as variables, from a particular section of code for troubleshooting purposes. Each statement is a macro that passes a set of arguments to the UST probe function.

Example 2 presents the source code for a sample instrumented application.

```
#include <stdlib.h>
#include <iostream>
#include <string>

using namespace std;

#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#include "com_ericsson_applicationx.h" // Must be included after the two #define statements.


int main ()
{
  int MyInt = 3;
  int i;
  int MyOverflow = 20;
 long array_long[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 char array_text[10] = "abcde";
  string MyString = "Still in the loop...";

  for (i=0; i<10; i++)
     {
       tracepoint(com_ericsson_applicationx, MyEventName1, i, MyInt);
       cout << "Hello World " << i << endl;
       tracepoint(com_ericsson_applicationx, MyEventName2, MyString.c_str());
       cout << ".........\n";
       sleep(2);
     }
  cout << "Program ends ! \n" ;
  tracepoint(com_ericsson_applicationx, MyEventName3, array_text, strlen(array_text),MyInt,MyOverflow);

}
```

Example 2 Sample Instrumented Application

This example highlights the elements that must be added to the application source code during instrumentation. Each element is described below.

Tracepoint statements use the following syntax:

```
tracepoint(tracepoint_provider, event_name, var1[,...,var10]);
```

Where:

tracepoint_provider

Is the trace domain that this event belongs to. To avoid name clashes, domain names must follow the trace naming conventions. For more information on the naming convention for tracepoint events, refer to the *Trace Event Guidelines* (Reference [1]).

Note: Each tracepoint provider must have a corresponding tracepoint provider header file that defines all of the tracepoint events in that domain. For more information on the tracepoint provider header file, refer to Section 3.1 on page 5.

event_name

Is the name of the event generated by this tracepoint statement. All event names for the same tracepoint provider must be unique.

Event names must follow the trace naming conventions. For more information on the naming convention for tracepoint events, refer to the *Trace Event Guidelines* (Reference [1]).

var1

Is the first of up to 10 arguments that can be passed to the UST probe function at runtime. All arguments must adhere to the following rules:

- Only primitive C-types, structures, and pointers are supported.

  `int`, `long`, `float`, `char`, `struct`, including arrays and pointers of these types.

  Note: For more information on using C structures in a tracepoint statement, refer to Section 7.1 on page 32.

- C++ classes, including strings, are not supported.

  A cpp-string (variable defined with a string class) must be converted to a null-terminated c-string before it can be passed as an argument to the UST probe function.

- When using a `ctf_sequence` or `ctf_sequence_text` macro to generate print output, two arguments are used for each array variable. The first argument specifies the array and the next argument carries the length of that array. For more information, see Section 3.1.6 on page 9.

To instrument your program code:

1. Define the `TRACEPOINT_DEFINE` and `TRACEPOINT_PROBE_DYNAMIC_LINKAGE` macros.

   ```
   #define TRACEPOINT_DEFINE

   #define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
   ```

   These macros allow your program to run independent of the LTTng-UST tracer. For more information on compiling an application binary that can run independent of LTTng libraries, refer to Section 5.2 on page 21.

2. Include the tracepoint provider header files in your program file.

   ```
   #include <tracepoint_provider_header_file>
   ...
   ```

These header files must contain declarations for all tracepoints used in your program. For more information on the tracepoint provider header files, refer to Section 3.1 on page 5.

3. Add tracepoint statements to the code, as needed.

   tracepoint(*<tracepoint_provider>*, *<event_name>*, var1 [,var_length] [,var2 ...]);

   Note: *var_length* is only required if the previous argument is an array and a `ctf_sequence` or `ctf_sequence_text` macro will be used to print the output.

   Example 3 highlights sample tracepoint statements:

```
int MyInt = 10;
tracepoint(APP1_domain1_branch1, event1, MyInt);
   // Variable MyInt is passed by value not by reference.

char my_CharArray[100]="hello";
tracepoint(APP1_domain1_branch2, event1, my_CharArray, strlen(my_CharArray));
   // Since the my_CharArray is an array, its length must be passed as
   // the next argument because ctf-sequence will be used in the
   // corresponding tracepoint provider header file.

long MyLongArray[]={1,2,3,4,5};
tracepoint(APP1_domain3, event3, MyLongArray, sizeof(MyLongArray)/sizeof(long));
   // Get the length of the array.

string My_cppString = “HELLO”;
tracepoint(APP1_domain2, event1, My_cppString.c_str());
   // Passing a c-string instead of the original. The array length is not needed
   // because ctf_string will be used in the corresponding tracepoint provider
   // header file.

tracepoint(APP1_domain1, event1, my_CharArray, strlen(my_CharArray), MyInt);
   // Multiple arguments are passed in one tracepoint.
```

Example 3 Sample Tracepoint Statements

4 Preparing the Tracepoint Probe

The third part of code instrumentation is the preparation of the tracepoint probe.

LTTng uses a probe function to retrieve runtime data from tracepoint statements and deliverer that data to the trace buffers. Tracepoint probes are application specific. These probes are automatically generated by the `TRACEPOINT_CREATE_PROBES` C macro using definitions from tracepoint provider header files.

As a C macro, `TRACEPOINT_CREATE_PROBES` must be defined in a .c file linked to your application that includes all of the tracepoint provider header files. Typically, the tracepoint probe is defined in a standalone file with the following format:

```
#define TRACEPOINT_CREATE_PROBES
#include "tracepoint_provider_1.h"
#include ...
...
```

Example 4 TRACEPOINT_CREATE_PROBES .c File

This file must be prepared prior to compiling your application and added to the Makefile.

5 Compiling Instrumented Applications

The following instructions document how to compile an instrumented application using Automake.

Building an instrumented application requires several modifications to your *Makefile.am*. These changes will vary depending on the application's intended runtime environment.

Typically, instrumented applications are compiled to be executable on systems where LTTng is installed; however, certain scenarios require an instrumented program to run on a target machine that does not have LTTng. In this case, the *Makefile.am* can be modified to build the application binary separately from the dependant LTTng libraries (the tracepoint probe). These dependencies are built into a separate library file. At execution, a shell wrapper can be used to preload (using LD_PRELOAD) the LTTng library file before calling the application binary.

Note: If an application is built with LTTng libraries linked in at compile time, it will only be executable if LTTng is installed on the target system.

If the LTTng library file is successfully preloaded (when LTTng is installed in the target machine), trace functionality can be applied to the instrumented application.

If the Lttng library file fails to preload, (when LTTng is not installed in the target machine), the instrumented application will still execute without trace functionality. There will be no performance impact on the instrumented application in this case.

This method ensures that the instrumented application can be executed regardless of the presence of LTTng libraries in the target machine.

5.1 Compiling the Tracepoint Probe Directly with the Application

Compiling the tracepoint probe directly with the application produces an application binary with all LTTng libraries linked in. This application will only be executable if LTTng is installed on the target system.

Prerequisites:

- To compile your instrumented application, all LTTng dependant files must be available on the development machine. These files are installed as part of the Trace EA SDK, which must be installed prior to building your application.
- Instrumented applications must be compiled using the DX GCC Cross-Compiler. For more information on the DX toolchain, refer to the documentation for your version of the product software.

Perform the following steps to update your *Makefile.am*:

1. Include the path to LTTng header files under `AM_CFLAGS` and `AM_CPPFLAGS`.

   For example:

   - **`AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include ⇒ -Wsystem-headers`**
   - **`AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include ⇒ -Wsystem-headers`**

   In this example, the variable `$DX_SYSROOT_X86_64` contains the Trace EA SDK installation path.

   Note: The `-Wsystem-headers` option ensures that the compilation will fail if any tracepoint provider name is longer than the 127 character maximum.

2. Specify your application source files.

   For example:

```
ApplicationX_SOURCES = applicationx.cpp applicationx_t
p.c
```

In this example, *applicationx.cpp* is the application source code and *applicationx_tp.c* defines the tracepoint probe. For more information on the tracepoint probe, refer to Section 4 on page 17.

3. Specify the LTTng libraries under `LDFLAGS`.

```
ApplicationX_LDFLAGS = -llttng-ust –llttng-ust-fork
```

Note: `llttng-ust` is the general LTTng library and `llttng-ust-fork` is a special LTTng library that allows the instrumented application make use of fork to spawn new processes that contain one or more tracepoints.

After performing these updates, your *Makefile.am* will appear similar to the following:

```
AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers
AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers

bin_PROGRAMS = ApplicationX
ApplicationX_SOURCES = ApplicationX.cpp  ApplicationX_tp.c
ApplicationX_LDFLAGS = -llttng-ust -llttng-ust-fork
```

Example 5 Sample Makefile.am

This makefile will generate a binary (ApplicationX) that has all of the LTTng dependant libraries linked-in. These dependencies can be viewed with the *ldd* command.

For example:

```
SC-2:/cluster/temp # ldd ApplicationX
linux-vdso.so.1 =>  (0x00007fff2f3ff000)
liblttng-ust.so.0 => /usr/lib64/liblttng-ust.so.0 (0x00007fea5bd47000)
liblttng-ust-fork.so.0 => /usr/lib64/liblttng-ust-fork.so.0 (0x00007fea5bb44000)
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fea5b839000)
libm.so.6 => /lib64/libm.so.6 (0x00007fea5b5e3000)
libc.so.6 => /lib64/libc.so.6 (0x00007fea5b285000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fea5b067000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007fea5ae63000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fea5ac4c000)
liblttng-ust-tracepoint.so.0 => /usr/lib64/liblttng-ust-tracepoint.so.0 (0x00007fea5aa45000)
librt.so.1 => /lib64/librt.so.1 (0x00007fea5a83c000)
liburcu-bp.so.1 => /usr/lib64/liburcu-bp.so.1 (0x00007fea5a636000)
libuuid.so.1 => /lib64/libuuid.so.1 (0x00007fea5a430000)
/lib64/ld-linux-x86-64.so.2 (0x00007fea5bf84000)
liburcu-common.so.1 => /usr/lib64/liburcu-common.so.1 (0x00007fea5a22d000)
```

Note: If any of the dependant libraries are not found at execution time, the program will not run.

5.2 Compiling the Tracepoint Probe Separately from the Application Using Dynamic Linking

Compiling the tracepoint probe separately from the application produces two files:

- The application binary.
- A separate library file (.so) that links in the tracepoint probe and all of the dependant LTTng files.

  Separating the tracepoint probe and LTTng dependant files from the application binary allows the instrumented application to operate independent of LTTng.

Prerequisites:

- To compile your instrumented application, all LTTng dependant files must be available on the development machine. These files are installed as part of the Trace EA SDK, which must be installed prior to building your application.
- Instrumented applications must be compiled using the DX GCC Cross-Compiler. For more information on the DX toolchain, refer to the documentation for your version of the product software.

Perform the following steps to update your *Makefile.am*:

1. Include the path to LTTng header files under `AM_CFLAGS` and `AM_CPPFLAGS`.

   For example:

   - **`AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include ⇒ -Wsystem-headers`**
   - **`AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include ⇒ -Wsystem-headers`**

   Note: The `-Wsystem-headers` option ensures that the compilation will fail if any tracepoint event name is longer than the 127 character maximum.

2. Specify your application source files.

   For example:

   `ApplicationX_SOURCES = applicationx.cpp`

Note: In this example, *applicationx.cpp* is the application source code. The tracepoint probe is not included in this step because it must be separated from the application source.

3. Specify that dynamically linked libraries will be used under `LDFLAGS`:

   `ApplicationX_LDFLAGS = -ldl`

4. Specify the creation of a separate library file (.so) that links in all of the LTTng dependant libraries.

   For example:

```
lib_LTLIBRARIES = my-lttng-libs-for-ApplicationX.la
my_lttng_libs_for_ApplicationX_la_SOURCES = ApplicationX_tp.c  \
                                            com_ericsson_applicationx.h

FORCE_SHARED_LIB_OPTIONS = -module -shared -avoid-version $(abs_builddir)
PROBE_LIBS = -llttng-ust -llttng-ust-fork
my_lttng_libs_for_ApplicationX_la_LDFLAGS = $(FORCE_SHARED_LIB_OPTIONS) $(PROBE_LIBS)
```

After performing these updates, your *Makefile.am* will appear similar to the following:

```
AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers
AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers

bin_PROGRAMS = ApplicationX
ApplicationX_SOURCES = ApplicationX.cpp
ApplicationX_LDFLAGS = -ldl

lib_LTLIBRARIES = my-lttng-libs-for-ApplicationX.la
my_lttng_libs_for_ApplicationX_la_SOURCES = ApplicationX_tp.c  \
                                            com_ericsson_applicationx.h

FORCE_SHARED_LIB_OPTIONS = -module -shared -avoid-version $(abs_builddir)
PROBE_LIBS = -llttng-ust -llttng-ust-fork
my_lttng_libs_for_ApplicationX_la_LDFLAGS = $(FORCE_SHARED_LIB_OPTIONS) $(PROBE_LIBS)
```

Example 6 Sample Makefile.am

To operate independently of LTTng , a shell wrapper must be used to run the instrumented application. This shell wrapper will attempt to preload the LTTng libraries before launching the application. If LTTng libraries fail to preload, (LTTng not available on the target machine) the application will still execute without trace functionality.

The shell wrapper for the sample application, *run_ApplicationX.sh*, is written as follows:

```
#!/bin/sh
LD_PRELOAD=/cluster/temp/my-lttng-libs-for-ApplicationX.so:liblttng-ust-fork.so.0.0.0⇒
 /cluster/temp/ApplicationX ${*}
```

Note: The "${*}" allows the passing of arguments to ApplicationX.

For example:

- To run ApplicationX directly:

```
./ApplicationX arg1 arg2
```

- To run with the shell wrapper:

```
./run_ApplicationX.sh arg1 arg2
```

When the above script is executed, it first tries to load the tracepoint probe that loads all of the dependent LTTng libraries. It then calls the application binary. If the dependant LTTng libraries are present, the application loads with tracing enabled. If the LTTng libraries are not present, the script will ignore the failure and launch the instrumented application with tracing disabled.

6 Instrumenting a Sample Application

As described in the previous sections, instrumenting an application is a four step process.

1. Adding tracepoint statements to the program code, see Section 6.2 on page 25.
2. Creating one or more tracepoint provider header files that define the tracepoint statements, see Section 6.3 on page 26.
3. Preparing the tracepoint probe, see Section 6.4 on page 30.
4. Modifying the *Makefile.am* to compile the instrumented application, see Section 6.5 on page 30.

The instrumentation process involves creating or modifying a number of files. These files are outlined in Figure 1.

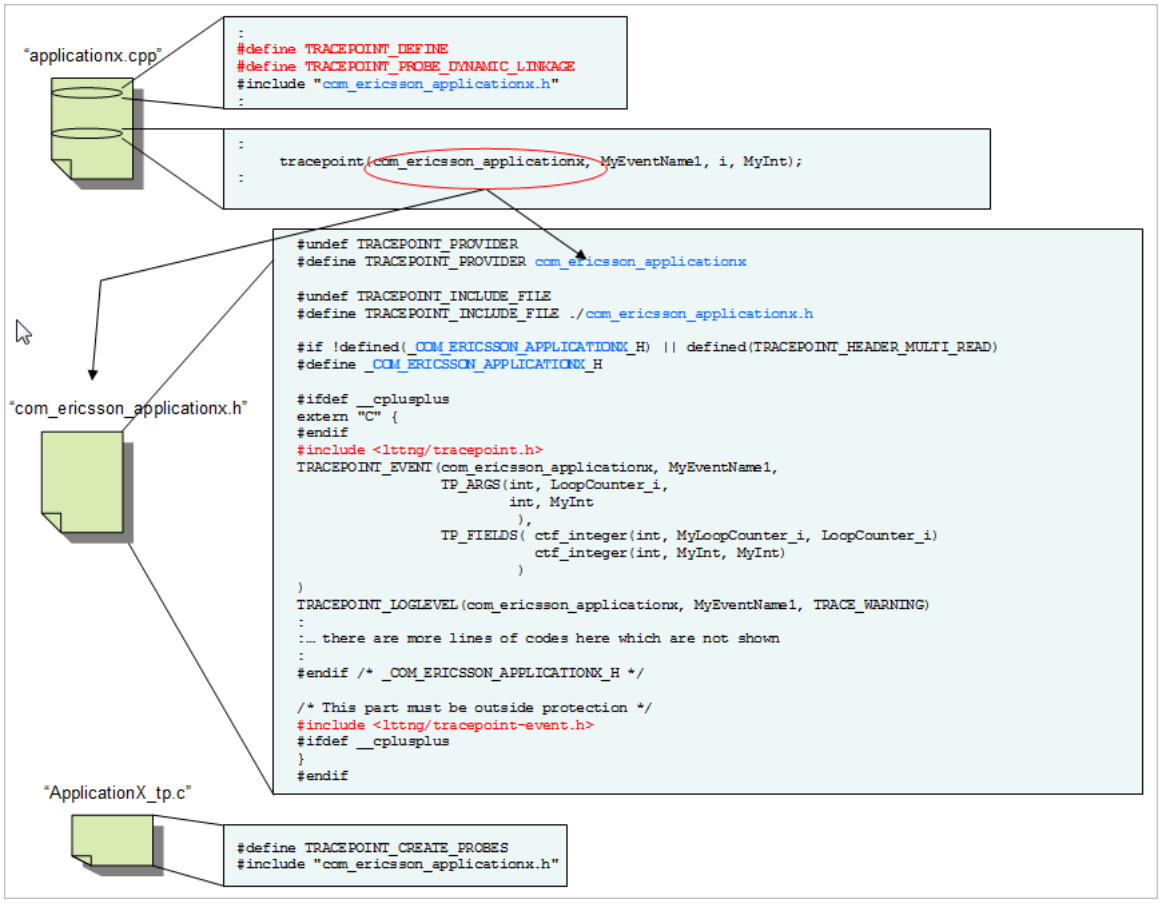

Figure 1 Instrumented Files

This section uses examples to present a walkthrough of the instrumentation process. Each example highlights the specific changes that are introduced to the code during that phase of the instrumentation.

6.1 The Sample Application

In this walkthrough we will be working to instrument a simple application called ApplicationX.

Example 7 shows the original ApplicationX source code.

```
#include <stdlib.h>
#include <iostream>
#include <string>

using namespace std;

int main ()
{
  int MyInt = 3;
  int i;
  int MyOverflow = 20;
  long array_long[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  char array_text[10] = "abcde";
  string MyString = "Still in the loop...";

  for (i=0; i<10; i++)
     {
       cout << "Hello World " << i << endl;
       cout << ".........\n";
     }
  cout << "Program ends ! \n" ;

}
```

Example 7 ApplicationX.cpp Before Instrumentation

ApplicationX is compiled using Automake with the following *Makefile.am*:

```
noinst_PROGRAMS = ApplicationX_notInstrumented
ApplicationX_notInstrumented_SOURCES = ApplicationX.cpp
```

Example 8 Makefile.am for ApplicationX

6.2 Adding Tracepoint Statements to the Program Code

The first step towards instrumenting ApplicationX is to add tracepoint statements to the program code. The specific changes are highlighted in the following example.

```
#include <stdlib.h>
#include <iostream>
#include <string>

using namespace std;

#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#include "com_ericsson_applicationx.h" // Must be included after the two #define statements.


int main ()
{
  int MyInt = 3;
  int i;
  int MyOverflow = 20;
 long array_long[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 char array_text[10] = "abcde";
  string MyString = "Still in the loop...";

  for (i=0; i<10; i++)
     {
       tracepoint(com_ericsson_applicationx, MyEventName1, i, MyInt);
       cout << "Hello World " << i << endl;
       tracepoint(com_ericsson_applicationx, MyEventName2, MyString.c_str());
       cout << ".........\n";
       sleep(2);
     }
  cout << "Program ends ! \n" ;
  tracepoint(com_ericsson_applicationx, MyEventName3, array_text, strlen(array_text),MyInt,MyOverflow);

}
```

Example 9 ApplicationX.cpp After Instrumentation

Trace output produced by ApplicationX is show in Section 6.6 on page 31.

Tracepoint statements are fully described in Section 3.2 on page 14.

Note: The `TRACEPOINT_DEFINE` and `TRACEPOINT_PROBE_DYNAMIC_LINKAGE` macros are defined to provide the option for the program to run independent of the LTTng-UST tracer.

`com_ericsson_applicationx.h` is the tracepoint provider header file that defines the new tracepoint statements. It is described in the following section.

6.3 Creating the Tracepoint Provider Header File

The second step towards instrumenting ApplicationX is to create a tracepoint provider header file that defines the tracepoint statements that were added to the source code.

A sample file is presented in the following example:

Note: Elements of the tracepoint provider header file are fully described in Section 3.1 on page 5.

```
#undef TRACEPOINT_PROVIDER
#define TRACEPOINT_PROVIDER com_ericsson_applicationx

#undef TRACEPOINT_INCLUDE_FILE
#define TRACEPOINT_INCLUDE_FILE ./com_ericsson_applicationx.h

#ifdef __cplusplus
extern "C" {
#endif

#if !defined(_COM_ERICSSON_APPLICATIONX_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _COM_ERICSSON_APPLICATIONX_H

#include <lttng/tracepoint.h>

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName1,
                 TP_ARGS(int, LoopCounter_i,
                         int, MyInt
                         ),
                 TP_FIELDS(
                           ctf_integer(int, MyLoopCounter_i, LoopCounter_i)
                           ctf_integer(int, MyInt, MyInt)
                         )
)

TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName1, TRACE_WARNING)

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName2,
                 TP_ARGS( const char *, MyString
                         ),
                 TP_FIELDS(ctf_string(string, MyString)
                         )
)
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName2, TRACE_WARNING)

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName3,
                 TP_ARGS( char *, array_text,
                          size_t, array_text_length,
                          int, MyNewLength,
                          int, MyOverflowLength
                          ),
                 TP_FIELDS(ctf_array(char, array, array_text, 3)
                           ctf_array(char, array2, array_text, 1)
                           ctf_array_text(char, arrayText, array_text, 2)
                           ctf_array_text(char, arrayText2, array_text, 20)
                           ctf_sequence(char, Sequence, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText2, array_text, size_t,MyNewLength)
                           ctf_sequence_text(char, sequenceText3, array_text, size_t,MyOverflowLength)
                           ctf_string(String, array_text)
                         )
)
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName3, TRACE_WARNING)


#endif /* _COM_ERICSSON_APPLICATIONX_H */

/* This part must be outside protection */
#include <lttng/tracepoint-event.h>

#ifdef __cplusplus
}
#endif
```

Example 10 com_ericsson_applicationx.h

Trace output produced by ApplicationX is show in Section 6.6 on page 31.

Note: Portions of the tracepoint provider header file can be generated by the `lttng-gen-tp` helper script. For more information, refer to Section 7.2 on page 34.

Each tracepoint statement in the program code must be defined as a separate TRACEPOINT_EVENT in the header file. ApplicationX has three events, highlighted in the following example.

```
TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName1,
                 TP_ARGS(int, LoopCounter_i,
                         int, MyInt
                          ),
                 TP_FIELDS(
                            ctf_integer(int, MyLoopCounter_i, LoopCounter_i)
                            ctf_integer(int, MyInt, MyInt)
                          )
)

TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName1, TRACE_WARNING)

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName2,
                 TP_ARGS( const char *, MyString
                          ),
                 TP_FIELDS(ctf_string(string, MyString)
                          )
)
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName2, TRACE_WARNING)

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName3,
                 TP_ARGS( char *, array_text,
                          size_t, array_text_length,
                          int, MyNewLength,
                          int, MyOverflowLength
                          ),
                 TP_FIELDS(ctf_array(char, array, array_text, 3)
                           ctf_array(char, array2, array_text, 1)
                           ctf_array_text(char, arrayText, array_text, 2)
                           ctf_array_text(char, arrayText2, array_text, 20)
                           ctf_sequence(char, Sequence, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText2, array_text, size_t,MyNewLength)
                           ctf_sequence_text(char, sequenceText3, array_text, size_t,MyOverflowLength)
                           ctf_string(String, array_text)
                          )
)
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName3, TRACE_WARNING)
```

Example 11 ApplicationX Tracepoint Events

The tracepoint events used in this example have been constructed to highlight specific features and functionality described below.

com_ericsson_applicationx:MyEventName1

The first event, com_ericsson_applicationx, MyEventName1, illustrates that parameters are passed by value in the tracepoint statement.

```
TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName1,
                 TP_ARGS(int, LoopCounter_i,
                         int, MyInt
                          ),
                 TP_FIELDS(
                            ctf_integer(int, MyLoopCounter_i, LoopCounter_i)
                            ctf_integer(int, MyInt, MyInt)
                          )
                )
```

In this example, a parameter i is passed in the tracepoint() statement from the main program. Here, TP_ARGS() receives the parameter with the

name `LoopCounter_i` and it is printed out by TP_FIELDS with the name `MyLoopCounter_i`.

This example demonstrates the flexibilty to modify parameter names at printout; nevertheless, it is strongly recommended to use the same parameter name passed in the tracepoint() statement. This best practice is highlighted in the second integer parameter `MyInt`.

com_ericsson_applicationx:MyEventName2

The second event, `com_ericsson_applicationx, MyEventName2`, illustrates the flexibility of using the `ctf_string` macro in TP_FIELDS for printing.

```
TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName2,
                 TP_ARGS( const char *, MyString
                        ),
                 TP_FIELDS(ctf_string(string, MyString)
                          )
                )
```

In this example, we do not need to pass the string length parameter of the string in TP_ARGS() because the `ctf_string` macro is being used to print it.

com_ericsson_applicationx:MyEventName3

The third event, `com_ericsson_applicationx, MyEventName3`, illustrates the other CTF printout macros that are supported by LTTng.

```
TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName3,
                 TP_ARGS( char *, array_text,
                          size_t, array_text_length,
                          int, MyNewLength,
                          int, MyOverflowLength
                          ),
                 TP_FIELDS(ctf_array(char, array, array_text, 3)
                           ctf_array(char, array2, array_text, 1)
                           ctf_array_text(char, arrayText, array_text, 2)
                           ctf_array_text(char, arrayText2, array_text, 20) // Note 20 > array_length
                           ctf_sequence(char, Sequence, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText, array_text, size_t,array_text_length)
                           ctf_sequence_text(char, sequenceText2, array_text, size_t,MyNewLength)
                           ctf_sequence_text(char, sequenceText3, array_text, size_t,MyOverflowLength)
                           ctf_string(String, array_text)
                          )
)
```

Example 12 MyEventName3

Note: More arguments can be passed to TP_FIELDS() than TP_ARGS().

6.4 Preparing the Tracepoint Probe

The third step towards instrumenting ApplicationX is to create a standalone *.c* file that defines the `TRACEPOINT_CREATE_PROBES` macro. This file must include all of the tracepoint provider header files that are used by the instrumented application. In this example, ApplicationX uses one tracepoint header file; therefore, the corresponding tracepoint probe file for this application contains the following:

```
#define TRACEPOINT_CREATE_PROBES
#include "com_ericsson_applicationx.h"
```

Example 13 ApplicationX_tp.c

Note: The filename of the tracepoint probe does not have to follow a specific naming convention, so long as it carries the *.c* extension.

6.5 Modifying the Makefile.am

The fourth step towards instrumenting ApplicatoinX is to modify the *Makefile.am* to compile the newly instrumented application.

There are two ways to modify the *Makefile.am*.

- To compile the tracepoint probe directly with the application.
- To compile the tracepoint probe separately from the application using dynamic linking.

Both methods are fully described in Section 5 on page 18.

The choice of method depends on application's intended runtime environment because compiling the tracepoint probe directly with the application creates a binary that is only executable on systems where LTTng is installed.

Compiling the Tracepoint Probe Directly with the Application

```
AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers
AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers

bin_PROGRAMS = ApplicationX
ApplicationX_SOURCES = ApplicationX.cpp  ApplicationX_tp.c
ApplicationX_LDFLAGS = -llttng-ust -llttng-ust-fork
```

Example 14 Makefile.am with LTTng Libraries Linked In

Compiling the Tracepoint Probe Separately from the Application Using Dynamic Linking

```
AM_CFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers
AM_CPPFLAGS = $(DX_SYSROOT_X86_64)/usr/include  -Wsystem-headers

bin_PROGRAMS = ApplicationX
ApplicationX_SOURCES = ApplicationX.cpp
ApplicationX_LDFLAGS = -ldl

lib_LTLIBRARIES = my-lttng-libs-for-ApplicationX.la
my_lttng_libs_for_ApplicationX_la_SOURCES = ApplicationX_tp.c  \
                                            com_ericsson_applicationx.h

FORCE_SHARED_LIB_OPTIONS = -module -shared -avoid-version $(abs_builddir)
PROBE_LIBS = -llttng-ust -llttng-ust-fork
my_lttng_libs_for_ApplicationX_la_LDFLAGS = $(FORCE_SHARED_LIB_OPTIONS) $(PROBE_LIBS)
```

Example 15 Makefile.am Separating LTTng Dependancies

This method builds the instrumented application without LTTng dependant libraries.

The makefile will generate the following output:

- `./ApplicationX` binary
- `.libs/my-lttng-libs-for-ApplicationX.so`

  The `.libs/my-lttng-libs-for-ApplicationX.so` file contains all of the LTTng library dependencies. If ApplicationX requires the LTTng libraries (when tracing is needed, for example), this file must be preloaded, using LD_PRELOAD, before launching ApplicationX. The preload is accomplished by launching ApplicationX through the following shell wrapper.

```
#!/bin/sh
LD_PRELOAD=/cluster/temp/my-lttng-libs-for-ApplicationX.so:liblttng-ust-fork.so.0.0.0⇒
 /cluster/temp/ApplicationX ${*}
```

Example 16 run_ApplicationX.sh Shell Wrapper

6.6 Trace Output

Activating all tracepoints in ApplicationX during program execution will generate the following trace output:

Note: The following example has been formatted for browser display.

```
 +- Time stamp hh:mm:ss.nano_second
 |
 |                    +- Process name of the instrumented application
 |                    |
 |                    |             +- PID of the instrumented application
 |                    |             |
 |                    |             |     +- Tracepoint name
 |                    |             |     |                                                    +- CPU Identifier (each node car
 |                    |             |     |                                                    |
 |                    |             |     |                                                    |                +- Tracepoint pa
 |                    |             |     |                                                    |                |
 V                    V             V     V                                                    V                V
[11:25:19.517394996] ApplicationX:8129 com_ericsson_applicationx:MyEventName1: { cpu_id = 6 }, { MyLoopCounter_i
[11:25:19.517438506] ApplicationX:8129 com_ericsson_applicationx:MyEventName2: { cpu_id = 6 }, { string = "Still
...
...
...
[11:25:37.518259500] ApplicationX:8129 com_ericsson_applicationx:MyEventName1: { cpu_id = 6 }, { MyLoopCounter_i
[11:25:37.518271861] ApplicationX:8129 com_ericsson_applicationx:MyEventName2: { cpu_id = 6 }, { string = "Still
[11:25:39.518353486] ApplicationX:8129 com_ericsson_applicationx:MyEventName3: { cpu_id = 6 }, { array = [ [0] =
```

Example 17 ApplicationX Trace Output

7 Appendix

7.1 Using C Structures in a Tracepoint Statement

There are two ways to pass a C structure in the tracepoint() function call:

1. Pass each structure element as a separate argument.

   For more information on passing arguments in the tracepoint() function call, refer to Section 3.2 on page 14.

   Note: A maximum of 10 arguments (apart from the provider name and event name) can be passed in the tracepoint() function call.

2. Pass a pointer to that structure as a single argument.

   The pointer can be used to access the structure elements for printout. This approach uses only one argument and allows the remaining nine to be used for other parameters.

   Note: There is no limit on the number of arguments (CTF macros) that can be used in TP_FIELDS to print output. For more information on TP_FIELDS, refer to Section 3.1.6 on page 9.

   To pass a pointer in the tracepoint function call, all structure members must be C-types. (`int`, `long`, `float`, and `char`). CPP-types (C-plusplus-types) are not allowed.

To avoid redefining the structure in the tracepoint provider header file, it is best to defined the structure in a separate header file. This header file must be included in the tracepoint probe.

The following examples illustrate how to pass a pointer to a structure in a tracepoint() function call.

Structure Header File

```
#ifdef __cplusplus
extern "C" {
#endif

struct subscriber {
     const char * name;     // can not use c-plusplus data types
     const char * imsi;
     float  balance;
     int secondRemaining;
};

typedef struct subscriber * p_subscriber;

#ifdef __cplusplus
}
#endif
```

Example 18 subscriber.h

Instrumented Application

```
#define TRACEPOINT_DEFINE
#include "subscriber.h"
#include "com_ericsson_common_trace_testapp.h"
...

p_subscriber get_p_subscriber(){
    static p_subscriber temp = (p_subscriber) malloc(sizeof( p_subscriber*));
    return temp;
}
...

p_subscriber personA = get_p_subscriber();
  personA->name = "Micky Mouse";
  personA->imsi = "123455143457900";
  personA->balance = 50.25;
  personA->secondRemaining = 3600;
...

tracepoint(com_ericsson_common_trace_testapp, pointer_to_structure_in_tracepoint, personA);
...
```

Example 19 InstrumentedApp.cpp

Tracepoint Provider Header File

```
...
TRACEPOINT_EVENT(com_ericsson_common_trace_testapp, pointer_to_structure_in_tracepoint,
            TP_ARGS(p_subscriber, personA
                    ),
            TP_FIELDS( ctf_string(Sub_Name, personA->name)
                       ctf_string(sub_IMSI, personA->imsi)
                       ctf_float(float, Sub_balance, personA->balance)
                       ctf_integer(int, Sub_TimeRemaining, personA->secondRemaining)
                     )
)
...
```

Example 20 com_ericsson_common_trace_testapp.h

Tracepoint Probe

```
#define TRACEPOINT_CREATE_PROBES
#include "subscriber.h"     // Must be included before
                            //  the associated header file.

#include "com_ericsson_common_trace_testapp.h
```

Example 21 tp.c

7.2 lttng-gen-tp Helper Script

To help users coding the tracepoint provider header files, LTTng has created a helper script that takes the `TRACEPOINT_EVENT` and `TRACEPOINT_LOGLEVEL` sections of the header file as input and outputs a complete tracepoint header file.

Note: `lttng-gen-tp` is not developed or maintained by Ericsson. The file is included in the Trace EA SDK as is to facilitate the instrumentation process.

`lttng-gen-tp` is located under `/<Trace EA SDK>/usr/bin/`.

Where *<Trace EA SDK>* represents the path at which the Trace EA SDK has been installed.

Prerequisites:

`lttng-gen-tp` requires a plain-text .template file as input. The template file must include all of the `TRACEPOINT_EVENT` and `TRACEPOINT_LOGLEVEL` definitions for the future header file.

Note: Each template file can only contain events for the same tracepoint provider.

`TRACEPOINT_EVENT` and `TRACEPOINT_LOGLEVEL` are fully described in Section 3.1 on page 5.

A sample template file is outlined in the following example:

```
TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName1,
                 TP_ARGS(int, LoopCounter_i,
                         int, MyInt
                          ),
                 TP_FIELDS(
                           ctf_integer(int, MyLoopCounter_i, LoopCounter_i)
                           ctf_integer(int, MyInt, MyInt)
                          )
)

TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName1, TRACE_WARNING)

TRACEPOINT_EVENT(com_ericsson_applicationx, MyEventName2,
                 TP_ARGS( const char *, MyString
                          ),
                 TP_FIELDS(ctf_string(string, MyString)
                          )
)
TRACEPOINT_LOGLEVEL(com_ericsson_applicationx, MyEventName2, TRACE_WARNING)
```

Example 22 Sample Template File

The filename of the template file should be the same as the corresponding tracepoint provider with a *.template* extension.

To use `lttng-gen-tp`:

- Launch `lttng-gen-tp` with your input file, specifying the output with the `-o` option.

  For example:

```
./lttng-gen-tp com_ericsson_applicationx.template -o com_ericsson_applicationx.h
```

A new tracepoint provider header file, *com_ericsson_applicationx.h*, is created using the `TRACEPOINT_EVENT` and `TRACEPOINT_LOGLEVEL` definitions specified in the template file, *com_ericsson_applicationx.template*.

Caution!

Risk of data corruption.

At the time of publication, `lttng-gen-tp` does not perform error checking. To ensure data integrity, you must manually ensure that the output from this tool complies with the instrumentation guidelines outlined in this document.

Reference List

[1] *Trace Event Guidelines*, 1/155 42-APR 901 0500/1

[2] *Trace EA Installation Instructions*, 1/1531-APR 901 0524/1