Technical Documentation

GPF

COMPRESSED/BINARY TRACING

Document Number:	89_03_16_00512
Version:	0.5
Status:	Final
Approval Authority:	
Creation Date:	2001-Oct-15
Last changed:	2015-Mar-08 by CKR
File Name:	89_03_16_00512_str2ind_userguide.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2001-Oct-15	CKR		0.1		
2001-Oct-29	C		0.2		
2002-Feb-25	CKR		0.3		
2003-Sep-11	CKR		0.4		
2015-Mar-08	CKR		0.5		

Table of Contents

List of Figures and Tables

List of References

[ISO 9000:2000]	International Organization for Standardization. Quality management systems - Fundamentals and vocabulary. December 2000

1 Introduction

This document describes concepts and tools used for the implementation of compressed/binary tracing.

Tracing is the process of assembling, transmitting and display of information on the current state of the protocol stack. Tracing is essential to the development and test of a protocol stack. Currently all traces are assembled, transmitted and displayed as character strings.

Especially on embedded targets, tracing raises a number of problems:

- The trace strings consume a considerable (possibly too large) amount of memory – ROM.
- The load on the (serial) interface of the target is too high, due to the large number of characters to be transmitted across this interface. This leads to either loss of some traces and/or effects on the real time behaviour of the protocol stack.

Tracing of the PC simulation on the other hand is not at all critical – no changes are needed.

The following chapters will address the problems above. First the current implementation of tracing will be analysed. Then a concept for the reduction of memory and interface load - compressed/binary tracing - will be introduced. After this a number of tools that help implementing compressed/binary tracing will be presented. Finally the integration of these tools into the build process and the interworking with the PCO will be explained.

2 Compressed/Binary Tracing

2.1 Current Implementation

For tracing a number of C Macros exist. In accordance to some of #define directives (NTRACE, TRACE_EVENT,...) these macros are translated into (trace) function calls. Among specialised functions e.g. for tracing operation codes (OPC), some general purpose functions exist. Most traces are realised with these functions. These trace functions, whose definitions are given below, are capable of tracing both static strings and run time formatted strings (vsprintf like).

```
int vsi_o_func_ttrace( char *format, ... );
int vsi_o_event_ttrace( char *format, ... );
int vsi_o_error_ttrace( char *format, ... );
int vsi_o_state_ttrace( char *format, ... );
int vsi_o_class_ttrace(ULONG trace_class, char *format, ... );
```

Due to their general nature and wide use within the protocol stack optimisation is concentrated on these functions. Let's have a look at two typical traces:

```
TRACE_FUNCTION("myFunction()");
TRACE_EVENT_P2("VarA %d VarB %d", a, b);
```

The C preprocessor would transform these traces into the following function calls:

```
vsi_o_func_ttrace("myFunction()");
vsi_o_event_ttrace("VarA %d VarB %d", a, b);
```

These functions will then transmit the strings via the (serial) interface.

2.2 New Concept

All strings solely used for tracing will be indexed. Instead of the storage and transmission of the complete string only the index is stored/transmitted. For run time formatted strings all additional parameters are transmitted too. The formatting (vsprintf) will be done at the PC side.

The index and additional parameters transmitted binary encoded i.e. four bytes for the index, and four/eighth bytes for each additional argument.

During the replacement of the strings a mapping table is maintained that will later be used for the retransformation from the index to the original string.

The two traces above would be transformed into the following function calls. These functions are special index trace functions solely implemented for this purpose:

```
vsi_o_func_itrace(17, “”);
vsi_o_func_itrace(18, “ii”, a, b);
```

The related entries in the mapping table would look as follows:

```
17,, myfunction()
18,ii, VarA %d VarB %d
```

To adopt this concept the original function call must be modified. Modification takes place between the phases of the pre-processor and the compiler. A tool will be run on the intermediate output of the pre-processor. It will do all the necessary modifications. The result is then fed to the compiler. This allows the source files to stay unmodified and thus normal tracing or no tracing is still possible.

The transmitted trace is marked with a special character as compressed/indexed. On the reception side for every marked trace the mapping table is used to reproduce the original string.

3 The Tools

3.1 str2ind

The tool str2ind transforms each call to any of trace functions mentioned above into a call to a corresponding index trace function. This index trace function differs in its name as well as in the number and kind of prototypes from the original trace function.

The tool expects the source file to be already pre-processed i.e. comments must have been removed, line continuation must have been resolved, trigraphs must have been expanded.

The tool does not only replace the function calls but it also maintains a mapping table that maps an index to the original string. For each replaced function call one entry is added to end of the mapping table. The index is incremented by one. Thus the mapping table growth incrementally.

str2ind works on up to 4 files:

- the source file (already pre-processed)
- the file that contains the mapping table
- the file that contains the version of the mapping table, this file will be compiled and linked to the protocol stack, so a runtime check whether the matching table is present at the displaying side is possible
- the log file
- the output file, which contains the modified function calls and serves as the input for the rest of the compilation, the name of the output file

The following is a description of the command line of the tool:

```
str2ind.exe [-a] [-d] -f file -t table [-l logfile] [-v versionfile] [-i 8|16|32|tms470]
```

```
file:          the name of the preprocessed source file
table:         the name of string mapping table
logfile:       the name of the logfile
versionfile:   the name of the version file
indexsize:     the max. size of the index. 8, 16 or 32 bit (default 32 bit).
               -i tms470 generates numbers suitable for compiler optimisation.

-a             append to an existing table - instead of writing a new one
-d             verbose information on screen
```

The tool str2ind produces an output file that has the same name as the original file with the character '_' added e.g. from myFile.pp the file myFile.pp_ would be generated.

Changes to the mapping table and to the version file only occur when a trace has been replaced. Otherwise these files stay untouched.

3.2 The mapping table

The mapping table is an ASCII text file. The first line contains the version of the table – the build date in seconds since UTC 1/1/70. The second line contains the largest index number used in this file. All other lines contain entries of the following form:

<index> <COMMA> < parameters> <COMMA> <string> <EOL>

The index increases continuously i.e. new entries are appended to the end of the table. Thus modified function calls get an index of the next unused index in the table. The field "parameters" of the entry contains the format string, that is transferred to index trace function. From it it's deducible how many additional parameters are transmitted and of what kind they are. The string is the original string.

A pseudo syntax of the file follows:

<version> <EOL>
<largest index used> <EOL>
<index> <COMMA> <parameters> <COMMA> <string> <EOL>
<index> <COMMA> <parameters> <COMMA> <string> <EOL>
<index> <COMMA> <parameters> <COMMA> <string> <EOL>
…
<largest index used> <COMMA> <parameters> <COMMA> <string> <EOL>
<EOF>

3.3 vsi_o_[func|error|event|state|class]_itrace()

These functions are implemented by the FRAME. They are responsible for the transmission of the trace:

```
SHORT vsi_o__func_ttrace(ULONG Index, char *format, …);
```

They expect the following arguments:

- Index: the index of the string
- format: a format string consisting of the characters: *,i,d,p,s
- …: additional arguments of any type
-

These functions analyse the format string. For each character in the string they expect one parameter on the call stack. The individual characters stand for certain argument types and are expected to be on the call stack and thus transmitted as follows:

- i: (Integer) 4 bytes
- d: (Double) 8 bytes
- p: (Pointer) 4 bytes
- *: (Integer) 4 bytes – vsprintf formatting information (width/precision)
- s: (String) n bytes – a string with a length that can't be calculated at compile time

All arguments are always transmitted binary coded in little endian byte order.

The real trace i.e. if you look at the bytes on the interface - is preceded by the character '%', which will then be used for the distinction of ordinary traces and compressed/indexed traces.

3.4 ind2str

This is a library that can be linked with any tool that wants to display compressed/binary traces. This library mainly consists of a function that retransforms an indexed string – marked by the character '%' – into the original string according to the mapping table.

This library also contains functions to query the version of a mapping table. This may be useful if more than one such table is present.

4 Usage and Integration

4.1 Usage

For developers of protocol stack entities changes become apparent. Still C macros are used to implement the traces. The syntax of these macros did not change.

One will notice the two more kinds of output files in the directory where the source files reside. These are the results of the transformations done by the pre-processor and the str2ind tool.

4.2 Integration into the build process

The integration into the build process can be quite simple. It only requires a C compiler that can be run as a pre-processor only e.g. TMS470.

The tool str2ind must be run between the pre-processor and the compiler. This requires a change in the Entity makefile.

The following command in a makefile:

```
$(CC) $^ $(CC_FLAGS)
```

should be replaced similarly:

```
$(CC) -po -p? $^ $(CC_FLAGS)
str2ind -a -l $(DIR)/str2ind.log -t $(DIR)/str2ind.tab -f $(addsuffix .pp,$(basename $^)) -v $(
DIR)/str2ind_v.c
$(CC) -c $(addsuffix .pp_,$(basename $^)) $(CC_FLAGS)
```

In the first step the preprocessor (TMS470) replaces the macros and produces an output file (myFile.pp). The preprocessor (-po) also removes comments and line continuation (\). In the preprocessor (TMS470) also replaces trigraphs (-p?). These transformations to the source file expected by str2ind. The call of the preprocessor may be different for a different compiler but should work as well.

In a second step the str2ind tool replaces all function calls and produces a new file (myFile.pp_). The mapping table (str2ind.tab) as well as the version file (str2ind_v.c) are updated if any trace are replaced. The run of the tool is recorded in a logfile (str2ind.log).

Finally the compiler is run on the former result producing an object file (myFile.obj).

In the top level makefile, the one that "makes" the protocol stack, a few changes are necessary too. The target clean should be extended by a command that removes the mapping table, the version file and the log file, e.g.

```
clean:
        ...
        $(RM) $(DIR)/str2ind.log $(DIR)/str2ind.tab
```

The target all must be extended by a number of commands. One that builds an object from the version file (str2ind_v.c) and another one that links the resulting object to the stack. This needs to be done, whenever a trace has been replaced.

Integrating the tool str2ind this way into the build process assures minimal rebuilds i.e. only from files that have been changed will be rebuild. The mapping table growths continuously, with entries for the replaced traces. At the same time its possible to start from the beginning with a "make clean".

For the PC simulation tracing stays as is i.e. all traces are transmitted as strings. When using makefiles it would be easily possible to similarly integrate str2ind.

4.3 Displaying the traces - PCO

The PCO is currently responsible for the display of the traces. To be capable of the display of indexed/compressed traces it must to be linked with the ind2str library.

It is crucial that the PCO uses the right version of the mapping table i.e. the one that corresponds to the build of the protocol stack.

At start up time, the protocol stack issues a string, which contains the version of the table it was build with.

The PCO loads the mapping table, formerly produced when building the stack. It can query the protocol stack for the version of the table with the SYSTEM primitive STR2INDVERSION. According to the answer it can check (possibly more than one table file) whether it uses the right table. This check can be performed automatically or manually.

When the PCO has located the right table it can reproduce the original traces. Otherwise indexed traces are displayed as an error message.

5 HowTo

Meanwhile integration of the tools took place for the project GPRS. This is described in the following chapter.

To use compressed tracing, only one flag in the g23.inf file must be set - **TRC = 4**. This flags conducts the whole build process to implement compressed tracing.

The created object directory and the library names, reflect the implementation of compressed tracing in their names. e.g. the generated libraries may be found in lib_gp_nf_**ct** instead of lib_gp_nf_**ct** and for example the ACI library may be named ac_sm_fd_tk_gp_pu_nf_**ct**.lib instead of ac_sm_fd_tk_gp_pu_nf_**tr**.lib. Also the name of the image reflects the use of compressed tracing in its name e.g. gsm_sm_gp_fd_ed_tk_pu_nf_**ct**_cs_cal_om_pa00_33_amd4_lj3.out.

At the toplevel directory (g23m) a new directory **trace** will be created during the build. Within this directory a subdirectory will be created for each distinct build configuration, i.e. whenever the build configuration is changed (via a change to g23.inf) a new directory will be created. This directory contains the trace mapping table (str2ind.tab), the log file (str2ind.log), the version file (str2ind.c) and a makefile(str2ind.mak). The name of this directory corresponds to the name of the image that will be build e.g. for the image gsm_sm_gp_fd_ed_tk_pu_nf_ct_cs_cal_om_pa00_33_amd4_lj3.out the trace directory will be trace\str2ind_sm_gp_fd_ed_tk_pu_nf_ct_cs_cal_om_pa00_33_amd4_lj3.

To enable PCO to decode the compressed traces. The mapping table(str2ind.tab) must be copied to the directory where PCO expects it. It's name must not be changed. In the future both the directory and the name of the file may be configurable.

Within the initialisation phase of the protocol stack, the stack reports the version of the mapping table that was created by the build of this stack within a special trace to the PCO. Upon reception of this version trace the PCO will check it against the version of the table it is currently using. Whenever these versions mismatch – an error trace is shown.

When connecting the PCO to an already running protocol stack, it is possible to query the stack of its version. This is done via the system primitive STR2INDVERSION. This primitive can be send from the PCO. Upon reception of the answer the will check the to version as described before.

Protocol stacks build using conventional tracing (TRC = 0/3) will report version 0. Traces are transmitted and displayed transparently.

6 Limitations

Currently escape sequences in trace strings e.g. \n \t \0x77, are not transformed into their binary encoding. They will be displayed as they a appear in the original string e.g. "Two \n lines " will not be a trace containing two lines, but a trace containing the two characters '\' and 'n'.