Programmer's Guide to the COND Facility

Generating and Reporting Conditions (Errors, Warnings)

Stephen M. Moore

Mallinckrodt Institute of Radiology
Electronic Radiology Laboratory
510 South Kingshighway Boulevard
St. Louis, Missouri 63110
314/362-6965 (Voice)
314/362-6971 (FAX)

Version 2.10.0
August 3, 1998

This document describes a reporting facility that may be used in programs to provide detailed information about conditions that may arise during execution.

1 Introduction

The COND facility is designed to maintain a stack of condition *values* and associated condition *messages*. A condition value is of type CONDITION and uniquely identifies the facility that generated the error, the severity of the error, and the actual error itself. A condition *message* is an ASCIIZ string (a 0 terminated character string) generated by a facility in association with the *value* and is designed to be readable by users or developers to determine the nature of the error. The combination of a value and a *message* is referred to as a *condition vector.*

Most functions are expected to return a condition *value*. When an exception or error occurs, the function is expected to push the <*value, message*> pair onto the stack and return the *value* to the caller. Functions are allowed to push multiple *vectors* on the stack to more fully specify the exception or error.

In the event of nested function calls, it is possible for a number of <*value, message*> pairs to be pushed onto the stack. An application could call facility A which in turn calls facility B. Facility B may push a *vector* onto the stack and return to facility A. Facility A senses the error, pushes its own *vector*, and returns its condition *value* to the application. The application can then examine the stack for detailed information about the failure.

It is assumed that most applications will only examine the value returned by the facility it called (facility A in the example above) and will not try to interpret secondary conditions on the stack. Applications usually dump the contents of the stack to some type of error log to be examined by developers.

2 Data Structures

2.1 CONDITION Data Type and Macros

As mentioned above, CONDITION is a data type defined in “condition.h” It contains three fields which define:

- facility
- severity
- value

Facility is a unique number assigned by an administrator that identifies a particular facility. In our system, we place these definitions in a central file that all modules are expected to include. For example, we define the following facilities for the DICOM project:

• FAC_DUL	1
• FAC_ACR	2
• FAC_IDX	3

Severity is one of a predefined set of values which indicates the severity of the error. These values are:

`SEV_SUCC`	A successful return value.
SEV_INFORM	An informational value.
SEV_WARN	The function was able to return successfully but some condition was detected.
SEV_ERROR	The function was not able to complete.
SEV_FATAL	The function was not able to complete and a fatal error occurred.

Value is assigned by the writer of the facility. It is a unique number that is tied to a particular condition.

The writer of the facility uses the macro FORM_COND (*facility, severity, value*) to create a legal CONDITION. This macro arranges the fields appropriately and generates a value with type CONDITION.

Each facility is expected to have at least one “normal” value which is the generic successful return value for the facility. This value should have the name FAC_NORMAL (for example, ACR_NORMAL). Other CONDITIONS are assigned as needed. These conditions are defined in the facility include file which is used by both the facility itself and by the application that wishes to use the facility.

Users of the facility can examine condition *values* for equality (if cond == ACR_X) or can test for the severity of a condition by using the following macros:

- `CTN_SUCCESS(x)`
- `CTN_CTN_INFORM(x)`
- `CTN_WARNING(x)`
- `CTN_ERROR(x)`
- `CTN_FATAL(x)`

These macros are used to hide the actual data representation from the user and will provide for more robust code. They are boolean expressions that return 1 if they are true (*x* is of severity `SUCCESS`) and 0 if not.

The macro `FACILITY(x)` will return the facility number associated with condition `x`. Thus, you could perform the following type of test:

```
if (FACILITY(x) == DUL_ACR)
```

2.2 Making a Condition Message

As discussed above, a condition *message* is an ASCIIZ string that provides a human-readable description of the condition. The *message* is formed by passing a control string and a variable number of arguments to `COND_PushCondition`. `COND_PushCondition` takes the control string and uses the `vsprintf` function to format the *message* that is placed on the stack. In this way run time messages can provide a little more information to the user (and to the developer). For example, you might call `COND_Push` Condition with the following arguments:

```
        XXX_FILEERR, “XXX - Failed to open file: %s”, fileName
```

(where `fileName` is a variable that holds the name of some file). `COND_PushCondition` will combine `“XXX - Failed to open file: %s”` with the filename and push the result on the stack with the condition *value*.

Note that we used an actual string in the example above. As a matter of practice, we choose to use macro constants for the control strings or create a function which returns a control string when given condition *value*. Either of these two practices will help make your facility a little more uniform so that the message for a particular condition looks the same if the message is generated in several different places in your facility.

3 Include Files

To use COND functions, applications need to include these files in the order given below:

```
#include “dicom.h”
#include “condition.h”
```

4 Return Values

This facility is different from the other facilities in that most routines return a value that is foreign to the facility.For example, the routine that pushes a *vector* onto the stack returns the value passed to it. The only return value defined for this facility is:

`COND_NORMAL`	Normal completion of a condition routine.

5 COND Routines

Detailed descriptions of the COND functions are included in this section.

Name

COND_CopyText - copy text from condition stack to caller's area

Synopsis

```
void CONDITION COND_CopyText(char *txt, size_t length)
```

txt	Pointer to memory in caller's address space to hold text written by this function.
length	Length of the txt buffer allocated by the caller.

Description

COND_CopyText is a function that provides a simple mechanism for extracting some of the text information from the condition stack. The caller allocates space in *txt* before calling this function. *COND_CopyText* will copy as much of the text information into the caller's area as allowed by *length*. The text from each condition on the stack is separated by '\n'.

Notes

The user may not like the format of the text returned or the fact that no condition values are written in the text. A more powerful function (*COND_ExtractConditions*) exists to give the user more control over format and amount of data.

Return Values

None

Name

COND_DumpConditons - dump the contents of the error stack.

Synopsis

```
void CONDITION COND_DumpConditions()
```

Description

COND_DumpConditions is a simple mechanism for dumping the error stack to *stderr*. This function prints each value and condition on the stack and then clears the stack.

Notes

Return Values

Name

COND_EstablishCallback - a user function to be called whenever COND_PushCondition is called.

Synopsis

```
CONDITION COND_EstablishCallback(void (*callback)())
```

callback	The function to be called whenever DUL_PushCondition is called.

Description

The function records the address of a callback routine to be called whenever DUL_PushCondition is called. The callback routine is called with arguments <value, message> which are to be pushed onto the stack. This function is a useful debugging tool which allows the application writer to log each condition as it occurs rather than waiting for a function to return. Writers of individual facilities would probably not call this function. To disable the callback, the function should be called with a NULL callback.

Return Values

```
COND_NORMAL
```

Name

COND_ExtractConditions - extract the (condition, message) pairs from the condition stack.

Synopsis

```
CONDITION COND_ExtractConditions(BOOLEAN (*callback)())
```

callback	The function to be called for each condition on the stack.

Description

This function examines each condition *vector* on the condition stack and calls the *callback* routine for each vector with arguments (value, message). The *callback* routine is a function that returns TRUE if more conditions are to be extracted from the stack and FALSE otherwise. This function does not alter the condition stack.

Return Values

```
COND_NORMAL
```

Name

COND_PopCondition - pop one or all conditions off the condition stack.

Synopsis

```
CONDITION COND_PopCondition(BOOLEAN clearStack)
```

clearStackBoolean	variable indicating if the entire stack is to be cleared.

Description

This function pops the top condition off the top of the stack if *clearstack* is FALSE and clears the entire stack if *clearstack* is TRUE. In either case, the function returns the CONDITION *value* that was on top of the stack.

If the stack is empty, the function returns COND_NORMAL.

Return Values

```
COND_NORMAL
```

Top *value* on the stack

Name

COND_PushCondition - push a (value, message) pair on the condition stack and return the condition value which was pushed.

Synopsis

```
CONDITION COND_PushCondition(CONDITION condition,
                             char*controlString, ...)
```

condition	The CONDITION value to be pushed on the stack.
controlString	An ASCIIZ string used as a control string for formatting the condition message.
...	Arguments as required by controlString.

Description

This function pushes a (condition, message) pair onto the control stack. The caller passes the condition to be placed on the stack and an ASCIIZ string which is a control string used by the standard C run time library *vsprintf* for formatting an output string. The caller passes optional arguments as required by the control string.

Notes

In the event that the stack overflows, all of the vectors on the stack are dumped to the standard error and the stack is reset.

Return Values

The condition value that was pushed onto the stack.

Name

COND_TopCondition - return the top (value, message) pair to the caller.

Synopsis

```
CONDITION COND_TopCondition(CONDITION *condition,
                            char *text, unsigned long maxLength)
```

condition	Caller’s variable to hold top condition on the stack.
text	Caller’s allocated area to hold text message from top condition.
maxLength	Maximum length of string to write into text.

Description

This function reads the top <*value, message*> pair from the top of the stack and returns the *value* to the caller. The caller provides storage access for the condition value and message and gives the length of the text area in the *maxlength* argument. The function writes the condition *value* and *message* into the caller’s allocated area and returns the top *value*. In the event that the stack is empty, the routine returns `COND_NORMAL`. The function does not alter the stack.

Return Values

`COND_NORMAL`
The top value on the stack.

Name

COND_WriteConditions - write the condition stack to a file

Synopsis

```
void CONDITION COND_WriteConditions(FILE *fp)
```

fp	File pointer for an existing (open) file. ASCII dump of condition stack will be written to this file.

Description

COND_WriteConditions is the proper implementation of *COND_DumpConditions*. It allows the caller to dump the condition stack to an arbitrary file that has been opened by the caller.

Notes

To write the stack to the stdout, `COND_WriteConditions(stdout);`

Once the stack is written to the file, it is cleared. Maybe that is not such a great side effect.

Return Values

None