Error Checking

INTRODUCTION

In every real-world application, a variety of errors might happen. Therefore, it is important for the system to be able to detect these errors and respond in an appropriate manner. In general, an RTOS has some error checking functionality built into it to detect errors and provide ways of responding to them to the application designer.

ERROR GROUPS

When developing and running RTOS applications, errors can be divided into two groups. The first group of errors is directed to the configuration of the RTOS and can be detected before compiling the application through sanity checks. These are called configuration checks.

When using an RTOS, it will usually be configured to meet the needs of the application through configuration values. These values might be altered by the application designer but must also comply to other constraints, like hardware or implementation constraints. The configuration checks are often done through preprocessor directives like #ifdef or #error in the RTOS code. The preprocessor examines the code before it is compiled and thus can check if there are any configuration errors.

A second group of possible errors are the ones that occur while the application is running and are therefore called runtime errors. There are many different reasons for why runtime errors occur and therefore different error detection strategies. This document concentrates on the second group of error checking.

RUNTIME ERRORS

Reasons for runtime errors include illegal parameters passed to API functions, like null pointers instead of valid task handles or values that are out of a given range. These errors can be detected through parameter checks inside the API functions before they are used.

Memory corruption can also lead to illegal or erroneous behavior and is another class of runtime errors. Memory checking algorithms can be used to detect these errors, like checksums or parity bits. If only a couple of very important values should be checked it is also possible to store copies (or bitwise inverted copies) of the values in a different memory location and to always compare them when accessing the value. These values are referred to as mirror values.

RTOS STACK

Each task in an RTOS has its own stack where local variables, return addresses, function parameters, return values and its context are stored. Therefore, the kernel must check that each stack resides in its own memory location, otherwise the content of the stack of one task may be corrupted by the other tasks. The size of the used stack grows with, for example, the depth of the function calls. As physical memory is finite, stack stack overflows may lead to corrupted memory. To prevent this error, the kernel of the RTOS must check the available space on the stack before the context of a task is stored.

FUNCTIONAL UNIT

Each task in an RTOS application is a functional unit on its own, with memory space associated with it. Access of a task to memory associated with another task may also lead to memory corruption and must be prevented. The memory protection unit, if available, can be used to define memory regions and access permissions for the tasks and will detect violations of them. If an task accesses memory which it has not been granted access to, the MPU will generate an exception which can be handled by the application.

ERROR HANDLING

Whenever an error is detected, it must be dealt with in some way by the application. How the error is signaled to the application depends on the type and severity of the error. One way to signal an error in an API function is through a status code in its return value. The application code should check the return value of all API functions. Depending on the error code, some errors might be recoverable by the application. In the case of an unrecoverable error, the application should transition the system into a safe state. Another way for the RTOS to signal errors is via a call to an application defined error handler function, which implements the behavior in case of an error. This method is often used for unrecoverable errors, where the error handler ends in an infinite loop so it never returns.

ERROR CHECKING & SAFE**RTOS**®

SAFE**RTOS** implements error checking of all parameters passed as API functions. The API functions of SAFE**RTOS** return a status code of type portBaseType which may contain error codes. All API functions return a status code which can be used to determine the success of the call. See file projdefs.h for a list of predefined error codes.

The error handler is called vApplicationErrorHook in SAFE**RTOS** and is a mandatory hook function which the application must define. It is called by the kernel if an error is detected and returning an error code is not possible.

Some examples of the error checks that are implemented by SAFE**RTOS** are:

- API function parameters are checked to hold valid handles and values.
- The stacks in SAFE**RTOS** are preallocated arrays of bytes that contain a stack-in-use marker to prevent it from being used by several tasks.
- The task control block contains information about the stack limit, so that kernel functions can check the available stack space before storing the tasks context on the stack and prevent stack overflows.
- Mirror values are used to check important values like the top of stack, the stack limit, using FPU marker and the MPU settings in the TCB. The tick value and the SVC handler address are also values that have a mirrored version stored.
- If the hardware contains an MPU, SAFE**RTOS** enables it by default and uses it to manage memory regions that the several tasks may access.

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

WITTENSTEIN