Result Library

Purpose

The `result` module defines a **shared result code system** for the codebase.

Its job is to give functions a consistent way to report success and failure without inventing random local conventions like:

- `0` means success here
- `1` means success there
- negative values somewhere else
- and one cursed module returning `true` for failure because someone was feeling creative

Instead, functions return a `result_t`, which makes error handling:

- consistent
- readable
- easier to propagate upward
- easier to log
- easier to document

This module also provides:

- string conversion helpers for result codes
- helper macros for early-return error propagation
- optional logging-aware propagation macros when the logging module is included

What files belong to this module

This module consists of:

- `result.h`
- `result.c`

`result.h`

Defines:

- the `result_t` enum
- the public string conversion functions
- the helper macros:
  - `TRY`
  - `TRY_CLEAN`
  - `TRY_LOG`
  - `TRY_LOG_CLEAN`

`result.c`

Implements:

- `result_to_short_str()`
- `result_to_desc_str()`

What problem this solves

In an embedded system, functions fail for many reasons:

- invalid arguments
- timeouts
- communication issues
- bad packet formats
- state machine misuse
- lack of memory
- busy resources
- and the usual parade of avoidable pain

> Without a shared result type, every module ends up inventing its own error style. This library creates one common language for reporting outcomes across modules.

That gives the codebase several benefits:

- function contracts are clearer
- errors can be passed up the call chain without translation
- logs can use the same human-readable error text
- helper macros reduce repetitive boilerplate

Core design

The design is intentionally simple:

- `RESULT_OK` means success
- every other `result_t` value represents a failure or exceptional condition
- functions return a single enum value
- callers decide whether to:
  - handle the error locally
  - return it upward
  - clean up before returning
  - log it before returning

This makes `result_t` a lightweight, shared error protocol.

Public API overview

The public API consists of:

- `result_t`
- `result_to_short_str()`
- `result_to_desc_str()`
- `TRY(expr)`
- `TRY_CLEAN(expr)`
- `TRY_LOG(expr)`
- `TRY_LOG_CLEAN(expr)`

String conversion functions

The module provides two functions for converting result codes into human-readable text.

These are useful for:

- logs
- diagnostics
- debug output
- CLI or terminal status messages
- test failure reporting

`result_to_short_str()`

```
const char *result_to_short_str(result_t code);
```

Purpose

Returns a short label for a result code.

Examples

- `RESULT_OK` -> `"OK"`
- `RESULT_ERR_TIMEOUT` -> `"Timeout"`
- `RESULT_ERR_INVALID_ARG` -> `"Invalid Argument"`

Default behavior

If the result code is unknown or unsupported, it returns:

```
"Unknown Error"
```

Intended use

This is best for compact output such as:

```
ERROR: Timeout
ERROR: Invalid Packet
ERROR: Buffer too small
```

`result_to_desc_str()`

```
const char *result_to_desc_str(result_t code);
```

Purpose

Returns a longer descriptive explanation of a result code.

Examples

- `RESULT_OK` -> `"The operation completed successfully."`
- `RESULT_ERR_TIMEOUT` -> `"An operation failed to complete within the allotted time."`
- `RESULT_ERR_INVALID_ARG` -> `"A provided argument is null, out of range, or otherwise invalid."`

Default behavior

If the code is unknown or unsupported, it returns:

```
"An unknown error code was encountered."
```

Intended use

This is useful when more context is needed, especially in logs:

```
Invalid Argument: A provided argument is null, out of range, or otherwise invalid.
```

Important implementation detail: mapping must stay synchronized

The enum in `result.h` and the switch statements in `result.c` must stay synchronized.

> At the moment, they are **not fully synchronized**.

Why this matters

This causes:

- misleading logs
- incomplete diagnostics
- confusion for anyone trying to use those result codes

Maintenance rule

Whenever a new `result_t` value is added, both conversion functions must be updated in the same change.

> **This should be treated as mandatory.**

Error propagation macros

The module provides a set of helper macros that reduce repetitive boilerplate when working with `result_t`.

These macros assume the common pattern:

- call a function returning `result_t`
- if it failed, stop current flow
- either return immediately or jump to cleanup

`TRY(expr)`

```
#define TRY(expr) \
  do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
      return _try_status; \
    } \
  } while (0)
```

Purpose

Evaluates an expression returning `result_t`.

If the result is not `RESULT_OK`, the current function immediately returns that result.

Example

```
result_t motor_start(void) {
  TRY(motor_check_ready());
  TRY(motor_enable_power());
  TRY(motor_configure_pwm());

  return RESULT_OK;
}
```

Expanded behavior

This behaves roughly like:

```
result_t status = motor_check_ready();
if (status != RESULT_OK) {
  return status;
}
```

for each call.

When to use it

Use `TRY()` when:

- the current function also returns `result_t`
- no local cleanup is needed before returning
- you want simple upward propagation

`TRY_CLEAN(expr)`

```
#define TRY_CLEAN(expr) \
  do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
      goto cleanup; \
    } \
  } while (0)
```

Purpose

Evaluates an expression returning `result_t`.

If the result is not `RESULT_OK`, execution jumps to a `cleanup:` label.

Example

```
result_t process_frame(void) {
  result_t status = RESULT_OK;
  void *buffer = NULL;

  buffer = malloc(128);
  if (buffer == NULL) {
    return RESULT_ERR_NO_MEM;
  }

  TRY_CLEAN(step_one());
  TRY_CLEAN(step_two());
  TRY_CLEAN(step_three());

  return RESULT_OK;

cleanup:
  free(buffer);
  return RESULT_ERR;
}
```

`TRY_LOG(expr)`

When `LOGGING_H` is defined, this macro becomes:

```
#define TRY_LOG(expr) \
  do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
      LOGE(TAG, "%s: %s", result_to_short_str(_try_status), \
           result_to_desc_str(_try_status)); \
      return _try_status; \
    } \
  } while (0)
```

Purpose

Like `TRY()`, but also emits a log message before returning.

Required assumption

The surrounding scope must define `TAG`, because the macro calls:

```
LOGE(TAG, ...)
```

If `TAG` is not defined, compilation will fail.

Example

```
#define TAG "NET"

result_t net_start(void) {
  TRY_LOG(net_hw_init());
  TRY_LOG(net_link_up());

  return RESULT_OK;
}
```

If `net_link_up()` returns `RESULT_ERR_TIMEOUT`, the log might look like:

```
[ERROR] NET: Timeout: An operation failed to complete within the allotted time.
```

and then the function returns that result.

`TRY_LOG_CLEAN(expr)`

When `LOGGING_H` is defined, this macro becomes:

```
#define TRY_LOG_CLEAN(expr) \
  do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
      LOGE(TAG, "%s: %s", result_to_short_str(_try_status), \
           result_to_desc_str(_try_status)); \
      goto cleanup; \
    } \
  } while (0)
```

Purpose

Like `TRY_CLEAN()`, but logs before jumping to `cleanup`.

Behavior when logging is not available

The logging-aware macros depend on whether `LOGGING_H` is defined.

This means they behave differently depending on whether the logging header has been included before `result.h`.

That is an important design detail.

When `LOGGING_H` is defined

If the logging header has already been included, `TRY_LOG` and `TRY_LOG_CLEAN` perform logging through `LOGE`.

This couples the macros to the logging module without hard-including it from `result.h`.

That keeps `result.h` lightweight, but also makes behavior depend on include order.

When `LOGGING_H` is not defined

The code falls back to compiler-specific warning behavior.

GCC / Clang

The macros emit a compile-time warning via `_Pragma(...)` and then degrade to:

- `TRY(expr)`
- `TRY_CLEAN(expr)`

MSVC

They emit a compiler message and also degrade to the non-logging versions.

Other compilers

A general `#warning` is emitted and the macros degrade to the non-logging versions.

Practical meaning

If logging is not available, the macros still work for flow control. They just do not log.

Recommended usage guidelines

Prefer specific result codes

Use the most precise `result_t` value that matches the failure.

Prefer:

- `RESULT_ERR_INVALID_ARG`
- `RESULT_ERR_TIMEOUT`
- `RESULT_ERR_NOT_INITIALIZED`

over generic:

- `RESULT_FAIL`

when possible.

Keep `RESULT_FAIL` as fallback only

`RESULT_FAIL` should mean:

> “something failed, but no existing specific code fits cleanly.”

It should not become the default.

Use `TRY()` only in functions returning `result_t`

Otherwise the generated `return _try_status;` is wrong.

Use `TRY_CLEAN()` only when a cleanup label exists

And only when you understand whether the error code is preserved.

Define `TAG` before using logging-aware macros

Without it, `TRY_LOG` and `TRY_LOG_CLEAN` are not valid.

Keep conversion functions updated

Whenever a new enum value is added, update:

- `result_to_short_str()`
- `result_to_desc_str()`

in the same commit.

This should be treated as mandatory maintenance.

Suggested mental model

Think of this module as:

> “The project-wide language for function outcomes.”

It is not just a list of enum values.

> It defines how modules communicate success and failure to each other, and the helper macros define the common patterns for passing those outcomes upward through the call stack.

That makes it foundational infrastructure, even if the code itself is small and visually innocent.

Revision #2
Created 2026-04-14 15:10:53 UTC by Nikolaos Diamantopoulos
Updated 2026-04-15 08:01:54 UTC by Nikolaos Diamantopoulos