Zephyr Workshop: Console, Logs & Shell

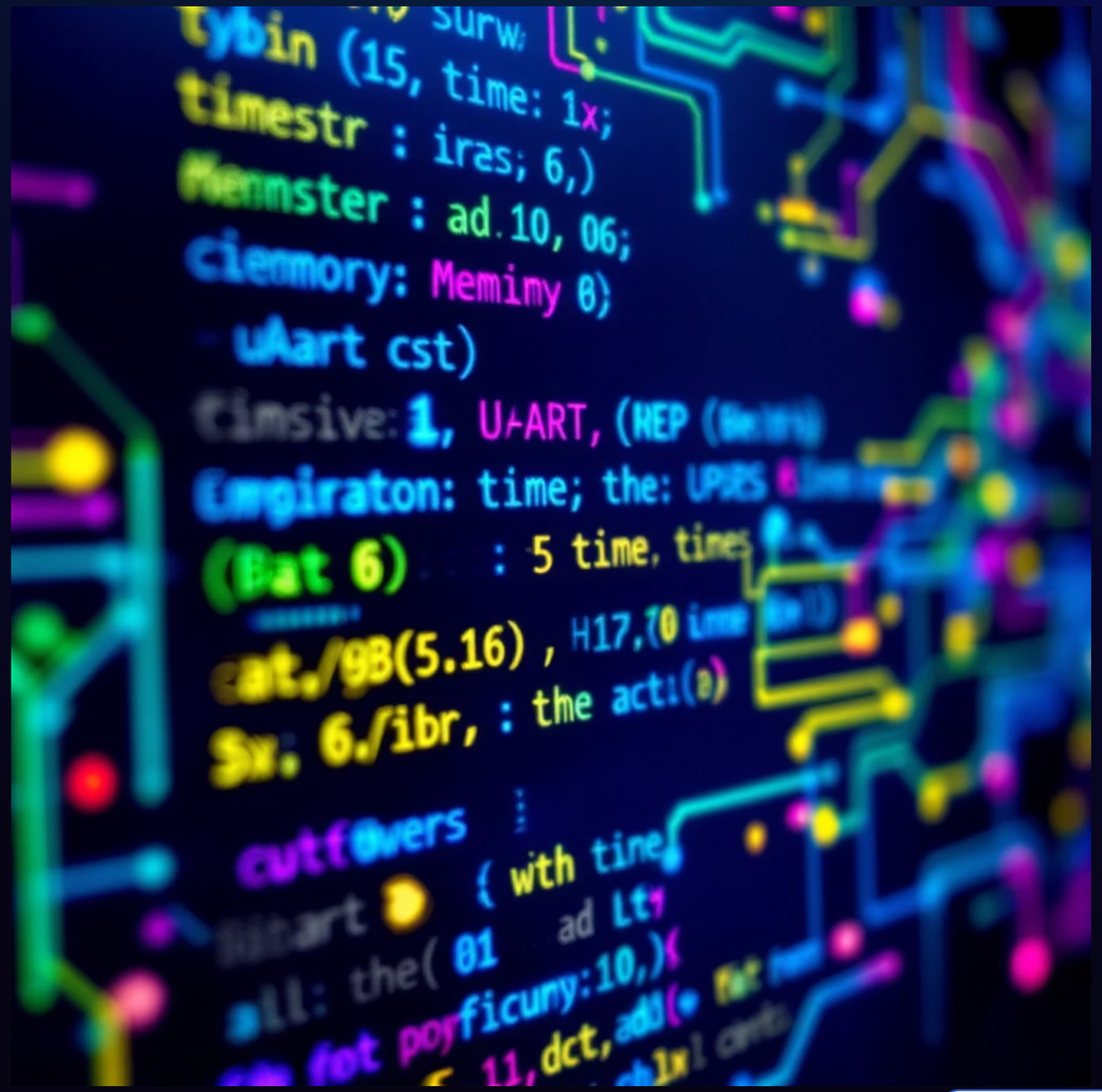
A hands-on guide to understanding output mechanisms, logging systems, and runtime interaction in the Zephyr RTOS environment

What is the Console in Zephyr?

The console subsystem in Zephyr provides an abstraction layer for where text output is sent from your application. By default, output is directed to a UART (serial port), but the system is flexible:

- UART (Universal Asynchronous Receiver/Transmitter) - default
- USB CDC (Communications Device Class)
- Dummy console for testing or systems without output
- Telnet for network-connected devices

Functions like `printk()` and the various logging macros utilize the console backend to display their output.



What is the Logging Subsystem?

Structured Reporting

Zephyr's logging subsystem provides a structured and consistent way to report messages throughout your application. This enables better organization and filtering of output.

Severity Levels

Messages can be categorized by importance:

LOG_ERR - Critical errors that prevent operation

LOG_WRN - Warning conditions requiring attention

LOG_INF - Informational messages about normal operation

LOG_DBG - Detailed debug information

Module-Based Configuration

Each module in your application can have its own log level, allowing for fine-grained control over verbosity. Output is directed to the configured console backend.

Logging Setup Example

Configuration

```
CONFIG_LOG=y
CONFIG_LOG_MODE_DEFERRED=y
CONFIG_LOG_BACKEND_UART=y
```

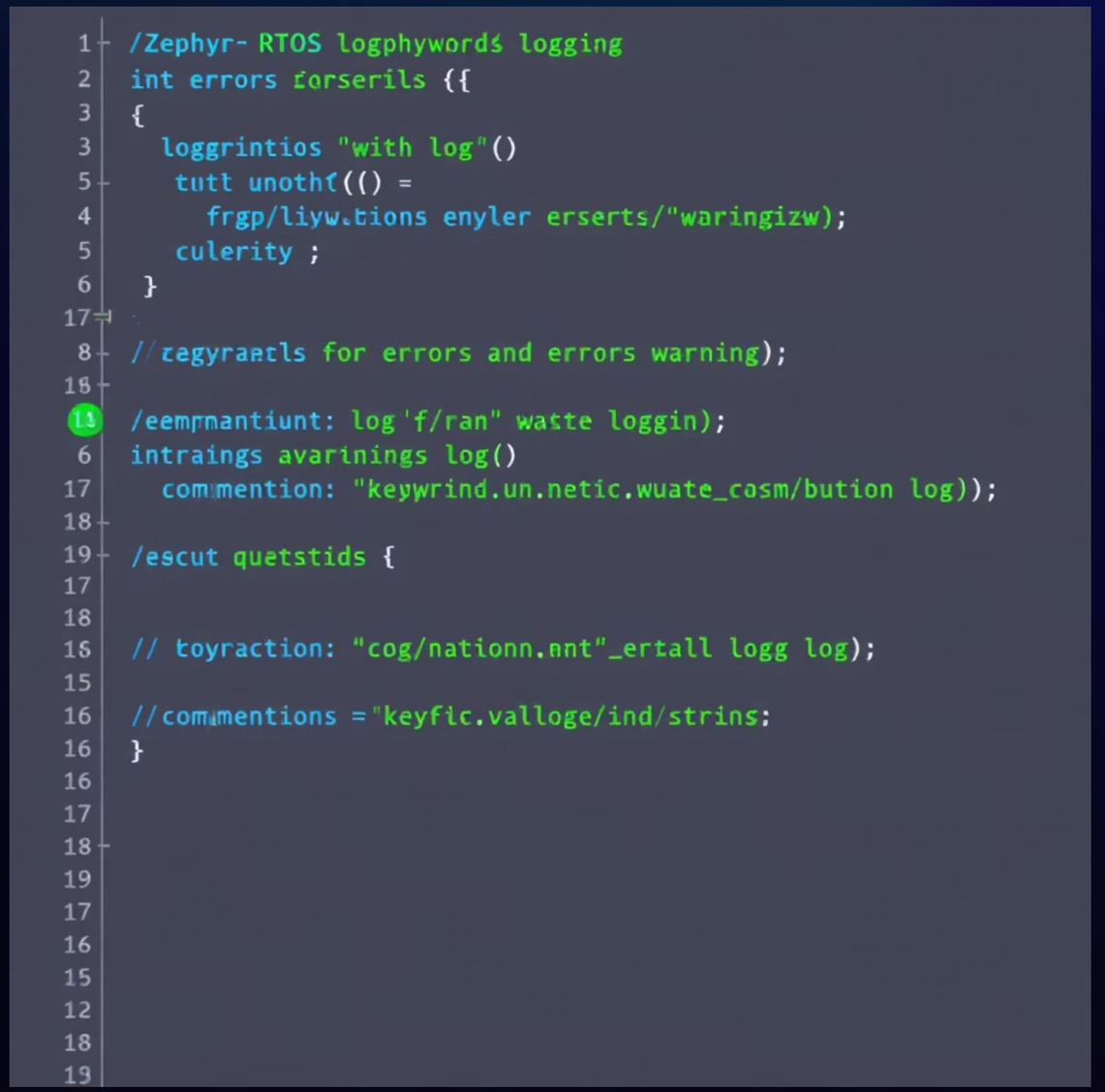
Code Implementation

```
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(tmp102sample);
```

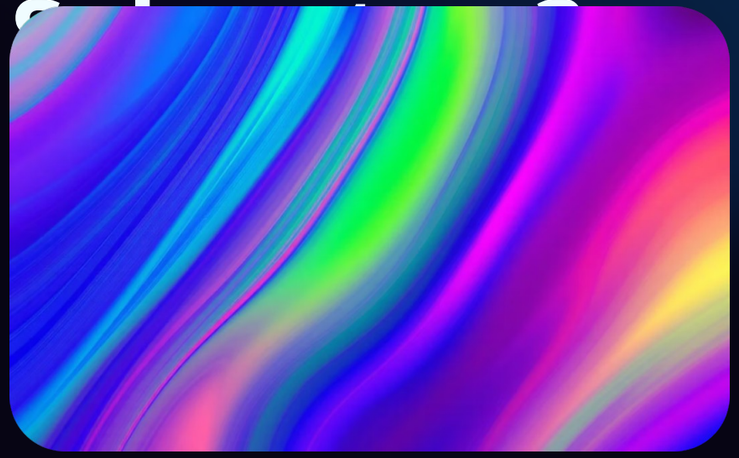
Replace standard print statements with logging macros:

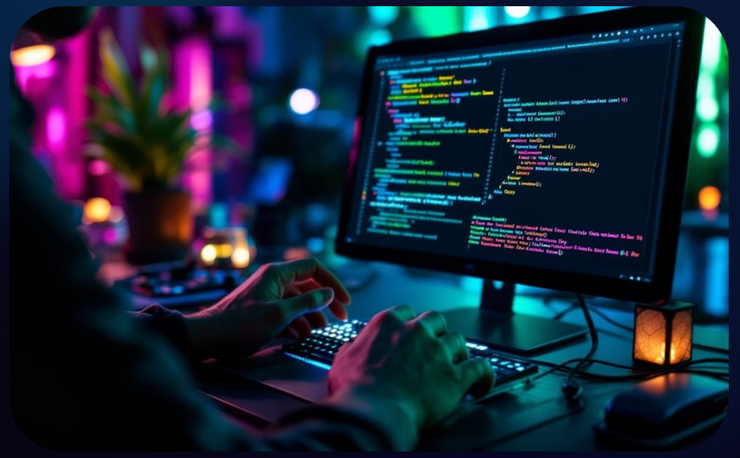
```
// Before: printk("Temperature: %d\n", temp);
// After:  LOG_INF("Temperature: %d", temp);
```

The deferred mode buffers log messages and processes them later, which is less intrusive to timing-sensitive code. The UART backend ensures messages appear on the serial console.



What is the Shell





Interactive Command-Line

The Shell subsystem provides an interactive command-line interface over UART or USB connections, allowing real-time control and monitoring of your application.

Runtime Control

Shell enables dynamic interaction with your application without recompilation. This is invaluable for debugging, testing, and runtime configuration adjustments.

Extensible Commands

Developers can leverage built-in commands or create custom commands specific to their application's needs, providing a powerful interface for system monitoring and control.

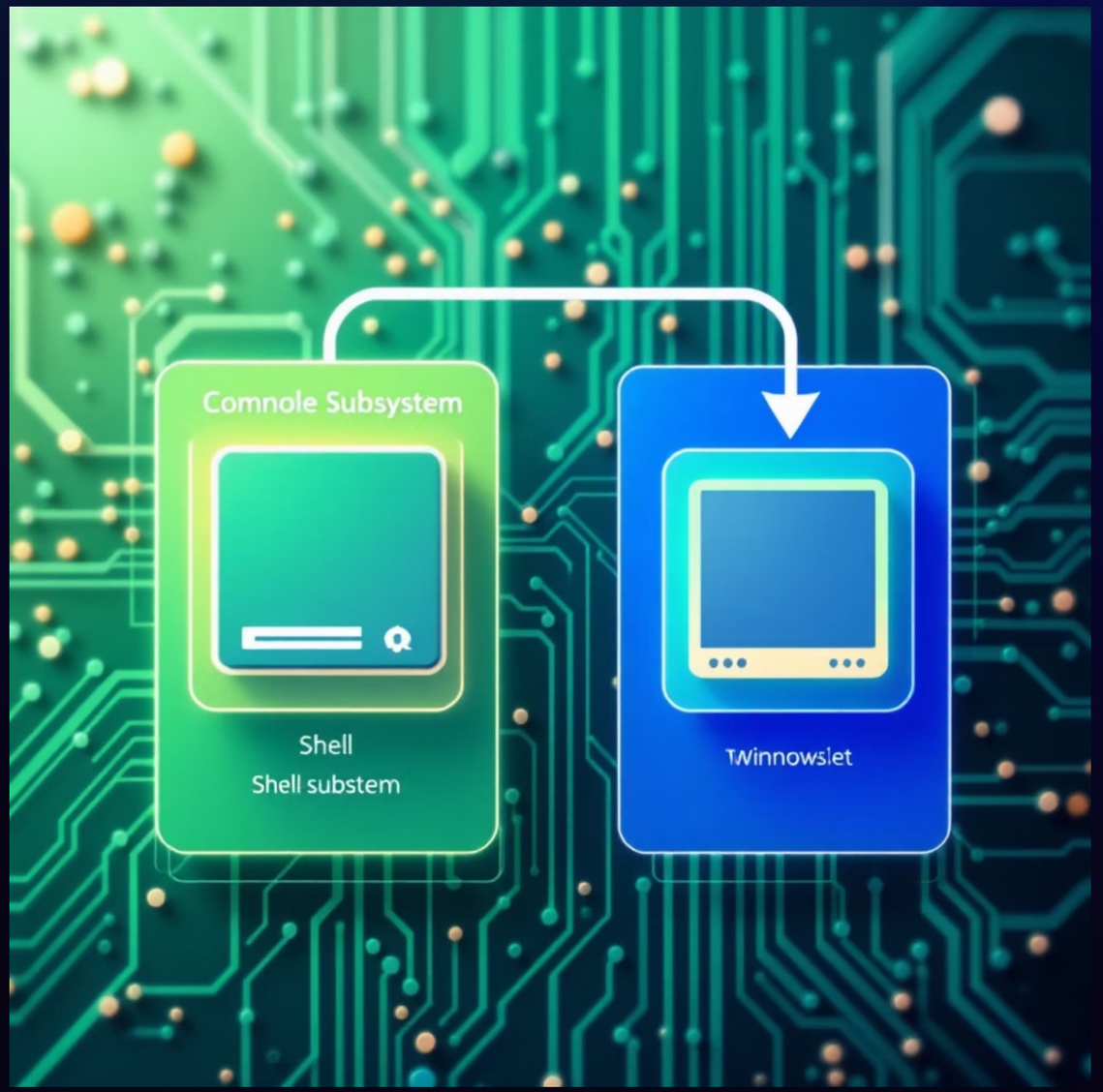
Shell Backend (Console Link)

The Shell subsystem uses the same backend infrastructure as the console, creating a consistent interface for both programmatic and interactive output:

- Shell commands and responses flow through the configured console backend

Most commonly used with `CONFIG_SHELL_BACKEND_SERIAL` for UART-based shells

- Can share the same physical interface as application logging
- Other option are BLE, USB, RTT, etc...



Shell Setup in

Enable Shell

```
CONFIG_SHELL=y
```

This activates the shell framework and core functionality, including command parsing, history, and tab completion.

Configure Console

```
CONFIG_UART_CONSOLE=y
```

Directs console output to the system's UART interface, typically connected to a serial terminal on your development computer.

Set Shell Backend

```
CONFIG_SHELL_BACKEND_SERIAL=y
```

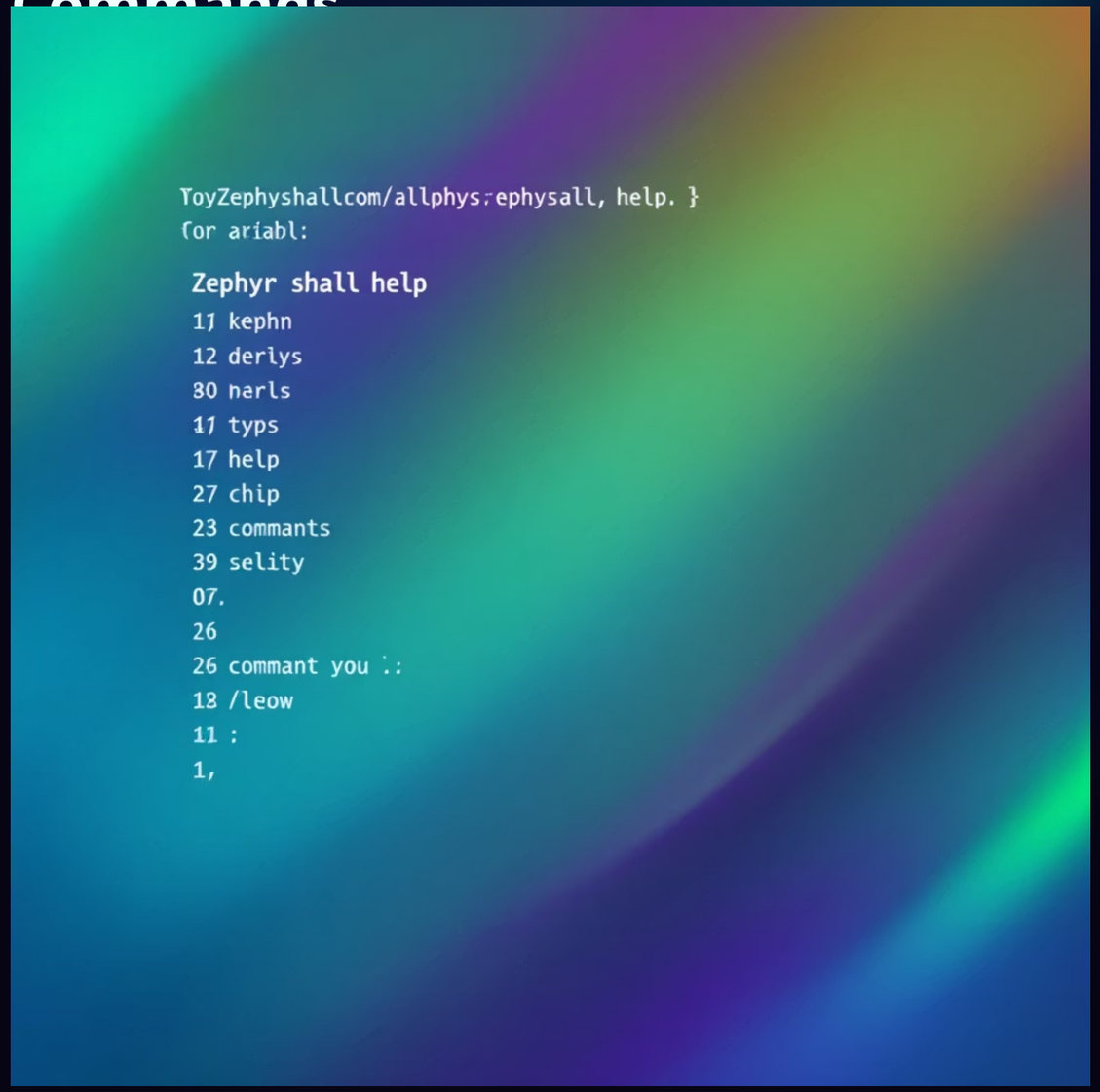
Configures the shell to use the serial (UART) backend for interactive command input and output.

Enable Basic Output

```
CONFIG_PRINTK=y  # Optional
```

Enables the `printk()` function for basic output messages through the console.

Built-in Shell Commands



Zephyr's shell comes with several useful built-in commands:

help	Lists all available commands or provides details on specific commands
kernel	Displays thread information, uptime, and other kernel statistics
device	Shows all initialized device drivers in the system
log	Controls and views logging levels for different modules
version	Shows the Zephyr version information
resize	Adjusts the terminal dimensions for proper formatting

Creating a Custom Shell Command

Adding your own commands to the shell is straightforward using the `SHELL_CMD_REGISTER()` macro:

```
static int cmd_my_command(const struct shell
*shell,                  size_t argc, char **argv)
{
shell_print(shell, "Command executed!");
return 0;
}
SHELL_CMD_REGISTER(mycommand, NULL,    "My custom
command description",    cmd_my_command);
```

The function signature requires:

- A pointer to the shell instance
- The argument count (argc)
- An array of argument strings (argv)

Commands can be organized in subcommand groups using `SHELL_STATIC_SUBCMD_SET_CREATE()`

Shell Command Examples

Study Existing Examples

Examine Zephyr's built-in command implementations in `kernel_service.c` and `sensor_shell.c` to understand best practices for command design.

The examples in these files demonstrate advanced features like parameter validation, subcommand handling, and formatting output for better user experience.

Live Activity

- Add shell command to your project (Sensor sample)

- Create `activate` command that will get an temperature as argument

    - If the current temperature is higher than the argument print "it's too hot"

- Run and verify over UART shell