\usepackage{fancyhdr} \pagestyle{fancy} \fancyhead[RE,RO]{\rightmark} \fancyfoot[C]{Page \thepage}

Chapter 1:

Baremetal IR OS Documentation

A Comprehensive Guide (v1.0.0)

Henrik Bach

June 2025

The Baremetal IR OS is an innovative operating system that runs directly on hardware without requiring a traditional host operating system. It is built around a custom Intermediate Representation (IR) that serves as the foundation for all code execution through a Just-In-Time (JIT) compilation engine.

1.1 1. Key Features

- **Hardware-level Integration**: Direct interaction with system hardware without an intermediary OS
- **Custom IR Architecture**: Platform-independent code representation for portability
- **JIT Compilation**: Dynamic code generation for optimal performance on various hardware
- **Minimal Runtime**: Efficient execution environment with minimal overhead
- **Extensible Design**: Modular architecture allowing for customization and expansion

1.1 2. Target Applications

- Embedded systems requiring precise control over hardware resources
- High-performance computing environments with specific optimization needs
- Research platforms for operating system design and implementation
- Educational tools for understanding low-level system operations

The development of the Baremetal IR OS was motivated by several key challenges in modern operating system design and implementation.

1.1 1. Traditional OS Limitations

Current operating systems face significant challenges:

1. **Performance Overhead**: Multiple abstraction layers introduce latency and resource consumption
2. **Hardware Compatibility**: Binary-level operating systems are tightly coupled to specific architectures
3. **Optimization Barriers**: Dynamic optimization is limited by predefined binary code structures
4. **Complexity**: Modern OS codebases have become increasingly complex and difficult to maintain

1.1 2. Our Solution Approach

The Baremetal IR OS addresses these limitations by:

1. **IR-Based Execution**: Using an intermediate representation allows for platform-agnostic code that can be optimized for any target hardware
2. **Direct Hardware Access**: Eliminating abstraction layers provides better performance and more precise control
3. **Dynamic Optimization**: The JIT compiler can apply optimizations specific to the current hardware and workload

4. **Simplified Architecture**: A clean-slate design focusing on essential functionality reduces complexity

The Baremetal IR OS uses a layered architecture with distinct components that work together to provide a complete operating system environment.

1.1 1. Core Components

1.1.1 1.1. 1. Hardware Abstraction Layer (HAL)

- Provides direct interfaces to hardware components
- Implements platform-specific drivers and controllers
- Exposes a uniform API for higher-level components

1.1.1 1.2. 2. IR Runtime Environment

- Manages IR code loading and execution
- Handles memory allocation for IR code and data
- Implements garbage collection and resource management

1.1.1 1.3. 3. JIT Compilation Engine

- Translates IR code to native machine code
- Performs optimization passes based on runtime information
- Manages code caching and recompilation

1.1.1 1.4. 4. OS Services

- Provides process and thread management
- Implements file system abstractions
- Handles inter-process communication
- Manages security and access control

1.1 2. System Flow

1. Boot sequence initializes hardware and core runtime
2. IR code is loaded from storage into memory
3. JIT compiler translates IR to optimized native code
4. Execution proceeds with dynamic optimization based on runtime conditions

Our custom IR serves as the foundation for all code execution in the Baremetal IR OS, providing a platform-independent representation that can be dynamically optimized.

1.1 1. IR Format Overview

The IR uses a hybrid design combining aspects of: - Static Single Assignment (SSA) form for data flow tracking - Control flow graphs for representing program structure - Type information for optimization and safety

1.1 2. Core IR Instructions

1.1.1 2.1. Memory Operations

- `load <type> <address> -> <result>`
- `store <type> <value> <address>`
- `alloc <type> <size> -> <result>`
- `free <address>`

1.1.1 2.2. Arithmetic Operations

- `add <type> <op1> <op2> -> <result>`
- `sub <type> <op1> <op2> -> <result>`
- `mul <type> <op1> <op2> -> <result>`
- `div <type> <op1> <op2> -> <result>`

1.1.1 2.3. Control Flow

- `branch <condition> <true_label> <false_label>`
- `jump <label>`
- `call <function> <args...> -> <result>`
- `return <value>`

1.1 3. Type System

The IR includes a comprehensive type system: - Primitive types (int32, int64, float32, float64, etc.) - Pointer types with metadata for memory management - Structure types for complex data organization - Function types with parameter and return information

1.1 4. IR Metadata

Instructions can include metadata for: - Optimization hints - Debug information - Safety checks - Hardware-specific directives

The Just-In-Time (JIT) compilation engine is responsible for translating our IR code into native machine code at runtime, with optimization tailored to the specific execution environment.

1.1 1. JIT Pipeline

The compilation process follows these stages:

1. **IR Loading**: Parse and validate IR code
2. **Analysis**: Gather statistics and identify optimization opportunities
3. **Optimization**: Apply IR-level transformations
4. **Code Generation**: Translate optimized IR to machine code
5. **Runtime Patching**: Update code based on execution data

1.1 2. Optimization Techniques

1.1.1 2.1. Static Optimizations

- Constant folding and propagation
- Dead code elimination
- Loop invariant code motion
- Strength reduction
- Inlining

1.1.1 2.2. Dynamic Optimizations

- Speculative execution
- Profile-guided optimization
- Type specialization
- Deoptimization for exceptional cases

1.1 3. Platform Adapters

The JIT engine includes pluggable backends for different architectures: - x86-64 - ARM64 - RISC-V - Custom hardware accelerators

1.1 4. Memory Management

- Code section allocation with proper permissions
- Inline cache for polymorphic operations
- Garbage collection integration
- Hot/cold code splitting

The Baremetal IR OS uses a specialized boot sequence to initialize hardware and establish the runtime environment for IR code execution.

1.1 1. Boot Sequence

1. **Firmware Handoff**: Receive control from platform firmware (UEFI/BIOS)
2. **Hardware Detection**: Identify and initialize essential hardware components
3. **Memory Setup**: Establish memory map and initialize memory management
4. **Core Runtime**: Load and initialize the IR interpreter and JIT compiler
5. **System Services**: Start essential system services
6. **Initial Application**: Load and execute the initial system application

1.1 2. Memory Layout

The system uses a carefully designed memory layout:

```
0x00000000 - 0x00FFFFFF: Reserved (Hardware, MMIO)
0x01000000 - 0x01FFFFFF: Boot code and data
0x02000000 - 0x0FFFFFFF: Kernel data structures
0x10000000 - 0x3FFFFFFF: JIT-compiled code cache
0x40000000 - 0x7FFFFFFF: Heap memory
0x80000000 - 0xFFFFFFFF: User application space
```

1.1 3. Runtime Services

During boot, the following runtime services are established:

- **Memory Manager**: Handles allocation, deallocation, and garbage collection
- **Thread Scheduler**: Manages execution of concurrent threads
- **IO Manager**: Provides abstracted interfaces for device I/O
- **Exception Handler**: Processes hardware and software exceptions
- **Security Monitor**: Enforces access control policies

1.1 4. Configuration Options

The boot process can be customized through:

- Command line parameters
- Boot configuration file
- Hardware-specific initialization modules

The Baremetal IR OS provides essential operating system services through specialized subsystems, all built on the core IR execution environment.

1.1 1. Process Management

- **Process Model**: Lightweight process containers with isolated memory spaces
- **Thread Management**: Cooperative and preemptive multithreading
- **Scheduling**: Priority-based scheduling with real-time support
- **IPC Mechanisms**: Shared memory, message passing, and synchronization primitives

1.1 2. Memory Management

- **Virtual Memory**: Paging with hardware acceleration when available
- **Memory Protection**: Process isolation and privileged access control
- **Allocation Strategies**: Custom allocators optimized for different use cases
- **Garbage Collection**: Optional GC for managed memory regions

1.1 3. File System

- **Virtual File System**: Unified interface for various storage backends
- **Native File Systems**: Custom file systems optimized for specific storage media
- **Caching**: Intelligent caching for improved I/O performance
- **Journaling**: Transaction support for data integrity

1.1 4. Networking

- **Protocol Stack**: Modular implementation of network protocols
- **Socket API**: Standard interface for network communication
- **Zero-copy I/O**: Efficient data transfer without unnecessary copying
- **Hardware Offloading**: Support for network hardware acceleration

1.1 5. Device Management

- **Driver Framework**: Structured API for device driver implementation
- **Device Discovery**: Automatic detection and configuration of hardware
- **I/O Scheduling**: Prioritization of device access requests
- **Power Management**: Device power state control for energy efficiency

1.1 6. Security

- **Access Control**: Fine-grained permission system
- **Memory Safety**: Runtime checks and isolation mechanisms
- **Secure Boot**: Verification of system integrity during startup
- **Encryption**: Built-in support for data encryption

The Baremetal IR OS interfaces directly with hardware components through a specialized Hardware Abstraction Layer (HAL) that provides both efficiency and portability.

1.1 1. Hardware Abstraction Layer

The HAL is structured in three tiers: 1. **Platform-Specific Layer**: Direct hardware access code for each supported platform 2. **Common Abstractions**: Unified interfaces for similar hardware components 3. **High-Level Services**: OS-level abstractions built on the lower layers

1.1 2. Supported Architectures

The system currently supports: - **x86-64**: Desktop and server systems - **ARM64**: Mobile and embedded devices - **RISC-V**: Open architecture platforms - **Custom Hardware**: FPGA-based accelerators and specialized processors

1.1 3. Driver Model

The driver architecture follows a modular design: - **Core Driver Framework**: Common infrastructure for all drivers - **Bus Drivers**: PCI, USB, I2C, SPI, etc. - **Device Drivers**: Storage, network, display, input, etc. - **Virtual Drivers**: Software-based device emulation

1.1 4. Hardware Acceleration

The system leverages hardware acceleration for: - **JIT Compilation**: Specialized instruction set extensions - **Memory Management**: Hardware page tables and TLBs - **Cryptography**: Hardware security modules - **Graphics**: GPU acceleration for rendering - **Networking**: Offloading packet processing to NICs

1.1 5. Hardware Configuration

Hardware resources are configured through: - **Static Configuration**: Pre-defined settings in system image - **Discovery**: Runtime detection of hardware capabilities - **Dynamic Configuration**: Adjustment based on workload requirements

This section provides guidance for developers working on the Baremetal IR OS project, including development environment setup, coding standards, and contribution guidelines.

1.1 1. Development Environment

1.1.1 1.1. Required Tools

- **Compiler Toolchain**: LLVM/Clang 15.0 or later
- **Build System**: CMake 3.21 or later
- **Emulation**: QEMU 7.0 or later for testing
- **Debugger**: GDB with target architecture support
- **Version Control**: Git 2.35 or later

1.1.1 1.2. Setup Instructions

```
# Clone repository with submodules
git clone --recursive https://github.com/baremetal-ir-os/core.git
cd core

# Install dependencies (Ubuntu/Debian)
./scripts/install-deps.sh

# Configure build
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Debug ..

# Build the project
make -j$(nproc)
```

1.1 2. Code Organization

The project follows a modular structure: - `src/core/` - Core runtime and JIT engine - `src/hal/` - Hardware abstraction layer - `src/os/` - OS services and subsystems - `src/ir/` - IR definition and utilities - `tools/` - Development and debugging tools - `tests/` - Test suites - `docs/` - Documentation

1.1 3. Coding Standards

- **Language**: Modern C++20 with limited use of platform-specific extensions
- **Style**: Follow the project style guide in `docs/style-guide.md`
- **Documentation**: All public APIs must be documented using Doxygen
- **Testing**: New features require unit tests and integration tests

1.1 4. Contribution Workflow

1. Create a feature branch from `develop`
2. Implement changes following coding standards

3. Add tests for new functionality
4. Submit a pull request with a detailed description
5. Address review feedback
6. Once approved, changes will be merged to `develop`

1.1 5. Common Development Tasks

1.1.1 5.1. Adding a New Hardware Platform

1. Create platform-specific HAL implementation in `src/hal/platforms/`
2. Implement required interfaces defined in `src/hal/hal_interfaces.h`
3. Add platform detection to the build system
4. Create platform-specific tests in `tests/platforms/`

1.1.1 5.2. Extending the IR Specification

1. Update IR definition in `src/ir/ir_spec.h`
2. Add validation rules to `src/ir/validator.cpp`
3. Implement interpretation in `src/core/interpreter.cpp`
4. Add code generation in `src/core/jit/codegen/`
5. Update documentation in `docs/ir-spec.md`

The Baremetal IR OS project employs comprehensive testing and debugging strategies to ensure reliability and facilitate development.

1.1 1. Testing Framework

1.1.1 1.1. Unit Testing

- Each component has dedicated unit tests
- Mocking framework for hardware dependencies
- Automated test runs on each commit

1.1.1 1.2. Integration Testing

- Full-system tests in emulated environments
- Hardware-in-the-loop testing for supported platforms
- Performance benchmarking suite

1.1.1 1.3. Test Organization

- `tests/unit/` - Component-level tests

- `tests/integration/` - System-level tests
- `tests/performance/` - Performance benchmarks
- `tests/platforms/` - Platform-specific tests

1.1 2. Running Tests

```
# Run all tests
cd build
make test

# Run specific test suite
ctest -R UnitTests

# Run with verbose output
ctest -V -R IntegrationTests
```

1.1 3. Debugging Tools

1.1.1 3.1. Trace and Logging

- Runtime configurable log levels
- Component-specific log channels
- Performance tracing infrastructure

1.1.1 3.2. Debugger Integration

- GDB server support for remote debugging
- JTAG interface for hardware debugging
- IR-level debugging with source mapping

1.1.1 3.3. Memory Analysis

- Memory leak detection
- Heap profiling
- Memory access validation

1.1 4. Debugging Process

1.1.1 4.1. System-Level Debugging

1. Enable verbose logging with `--log-level=debug`
2. Capture boot sequence with `--trace-boot`
3. Use the integrated debugger with `--debug-port=1234`

4. Connect with GDB: `gdb -ex "target remote localhost:1234"`

1.1.1 4.2. IR Debugging

1. Compile with debug info: `--ir-debug-info`
2. Use the IR debugger: `ir-debug program.ir`
3. Set breakpoints on IR instructions
4. Inspect IR state during execution

1.1.1 4.3. JIT Debugging

1. Enable JIT debugging with `--jit-debug`
2. Dump generated code with `--dump-jit-code=file.asm`
3. Use the JIT profiler with `--jit-profile`
4. Analyze hotspots with `analyze-jit-profile results.prof`

1.1 5. Issue Reporting

When reporting issues, please include: 1. Detailed description of the problem 2. Steps to reproduce 3. System configuration 4. Relevant logs and debug output 5. Expected vs. actual behavior

This roadmap outlines the planned development trajectory for the Baremetal IR OS project over the next several release cycles.

1.1 1. Current Version (1.0.0)

The initial release includes: - Core IR specification and implementation - Basic JIT compiler for x86-64 and ARM64 - Fundamental OS services (memory, threads, basic I/O) - Developer documentation and tools - Support for common development boards

1.1 2. Short-Term Goals (1.x)

1.1.1 2.1. Version 1.1 (Q3 2025)

- Improved memory management with generational GC
- Extended device driver framework
- Initial network stack implementation
- Performance optimizations for JIT compiler
- Enhanced debugging tools

1.1.1 2.2. Version 1.2 (Q4 2025)

- File system enhancements with journaling
- RISC-V architecture support
- Security hardening features
- Inter-process communication improvements
- Extended standard library

1.1.1 2.3. Version 1.3 (Q1 2026)

- Graphics and display subsystem
- USB device support
- Power management framework
- Initial real-time scheduling capabilities
- Extended tooling ecosystem

1.1 3. Medium-Term Goals (2.x)

1.1.1 3.1. Version 2.0 (Q3 2026)

- Complete IR optimization framework
- Advanced security model with formal verification
- Distributed computing capabilities
- Hardware acceleration for critical paths
- Application ecosystem expansion

1.1.1 3.2. Version 2.x Features

- Virtualization support
- Cloud integration capabilities
- Machine learning acceleration
- Self-optimizing runtime
- Expanded hardware platform support

1.1 4. Long-Term Vision (3.x and beyond)

- Formal verification of critical system components
- Hardware co-design opportunities
- Specialized versions for IoT, edge, and high-performance computing
- Advanced autonomic computing features
- Academic and industry partnership programs

1.1 5. Contributing to the Roadmap

We welcome community input on our development priorities. To contribute: 1. Join our community discussions on Discord or the mailing list 2. Submit feature proposals via GitHub issues 3. Participate in our quarterly roadmap planning sessions 4. Contribute proof-of-concept implementations

1.1 6. Release Schedule

Version	Expected Date	Focus Areas
1.0.0	June 2025	Initial stable release
1.1.0	September 2025	Performance and networking
1.2.0	December 2025	File systems and new architectures
1.3.0	March 2026	User interfaces and devices
2.0.0	September 2026	Major architectural enhancements

This document provides a detailed specification of the Intermediate Representation (IR) used in the Baremetal IR OS.

1.1 1. IR File Format

IR code is stored in text format with the following structure:

```
; Module declaration
module "example_module"

; External declarations
external func @printf(i8*, ...) -> i32

; Global variables
global @counter i32 = 0

; Type definitions
type %person = { i8*, i32 }

; Function definition
func @main() -> i32 {
  ; Basic blocks
  entry:
    %ptr = alloc i32
    store i32 42, %ptr
    %val = load i32, %ptr
    br %val, eq, 42, then, else

  then:
    %result = call @compute(i32 %val)
    return i32 %result

  else:
    return i32 0
}
```

1.1 2. Type System

1.1.1 2.1. Primitive Types

- `i8`, `i16`, `i32`, `i64`: Integer types of specified bit width
- `u8`, `u16`, `u32`, `u64`: Unsigned integer types
- `f32`, `f64`: Floating-point types
- `bool`: Boolean type (1-bit)
- `void`: Represents no value

1.1.1 2.2. Derived Types

- Pointers: `<type>*` (e.g., `i32*`)
- Arrays: `[<size> x <type>]` (e.g., `[10 x i32]`)
- Structures: `{ <type>, <type>, ... }` (e.g., `{ i32, i8* }`)
- Functions: `(<param_types>) -> <return_type>`

1.1 3. Instructions

1.1.1 3.1. Memory Operations

- `%ptr = alloc <type> [, <size>]`: Allocate memory
- `free <ptr>`: Free allocated memory
- `%val = load <type>, <ptr>`: Load value from memory
- `store <type> <val>, <ptr>`: Store value to memory
- `%ptr = getelementptr <type>, <ptr>, <indices...>`: Compute address of structure element

1.1.1 3.2. Arithmetic Operations

- `%result = add <type> <op1>, <op2>`: Addition
- `%result = sub <type> <op1>, <op2>`: Subtraction
- `%result = mul <type> <op1>, <op2>`: Multiplication
- `%result = div <type> <op1>, <op2>`: Division
- `%result = rem <type> <op1>, <op2>`: Remainder
- `%result = neg <type> <op>`: Negation

1.1.1 3.3. Bitwise Operations

- `%result = and <type> <op1>, <op2>`: Bitwise AND
- `%result = or <type> <op1>, <op2>`: Bitwise OR
- `%result = xor <type> <op1>, <op2>`: Bitwise XOR
- `%result = shl <type> <op>, <bits>`: Shift left
- `%result = shr <type> <op>, <bits>`: Shift right
- `%result = not <type> <op>`: Bitwise NOT

1.1.1 3.4. Control Flow

- `br <cond>, <op>, <val>, <true_label>, <false_label>`: Conditional branch
- `jump <label>`: Unconditional jump
- `%result = call <func>(<args>...)`: Function call
- `return <type> <value>`: Return from function
- `unreachable`: Marks unreachable code

1.1.1 3.5. Conversion Operations

- `%result = zext <from_type> <value> to <to_type>`: Zero extension
- `%result = sext <from_type> <value> to <to_type>`: Sign extension
- `%result = trunc <from_type> <value> to <to_type>`: Truncation
- `%result = bitcast <from_type> <value> to <to_type>`: Bit pattern reinterpretation
- `%result = inttoptr <int_type> <value> to <ptr_type>`: Integer to pointer conversion
- `%result = ptrtoint <ptr_type> <value> to <int_type>`: Pointer to integer conversion

1.1 4. Metadata

Instructions and declarations can include metadata:

```
%result = add i32 %a, %b, !debug !1, !optimize !2

!1 = !{!"line", 42, "file.c"}
!2 = !{!"inline"}
```

1.1 5. Extensions

The IR supports extensions for specialized hardware:

```
; Vector operations
%vec = vload <4 x f32>, %ptr
%result = vadd <4 x f32> %vec1, %vec2

; Custom hardware accelerator
%result = hwaccel "crypto.aes", i8* %data, i64 %len, i8* %key
```

1.1 6. Binary Format

For efficient storage and transmission, the IR can be serialized to a binary format (BIRF) with the following sections:

1. Header: Magic number, version, module name
2. Type Table: Type definitions
3. Global Variables: Global variable declarations
4. Function Table: Function declarations and signatures
5. Instruction Stream: Serialized instruction sequences
6. Metadata Table: Associated metadata
7. String Table: String constants