COMPUTARIUM
Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

Microcontroller Programming Project

computarium.lcd.lu presents

CERES 1.0: A new language for Harvard Architecture MCUs

PART I: project description

Author:
claude.baumann@education.lu

This project has primarily educational objectives
May 20, 2025

Abstract

This work presents an educational approach rooted in constructivist learning theory, emphasizing experiential learning to deepen understanding of embedded systems programming. Central to the project is the design and implementation of a recursive compiler for a custom programming language tailored specifically for microcontroller (MCU) firmware development. Unlike traditional methods that depend on high-level languages such as C or C++, this initiative gives learners direct control over the generated machine code, operating at the bare-metal level. This approach integrates principles of data safety and malware prevention by promoting minimal runtime environments, clear memory management semantics, and deterministic behavior—key factors in secure embedded development. Through the end-to-end creation of both language and compiler, students gain practical insights into language design, parsing, code generation, and hardware-level programming. This hands-on methodology not only addresses the limitations of conventional toolchains in embedded contexts but also fosters a deeper understanding of system-level computing, security, and software reliability through active creation and experimentation.

History:

- Version 1.0: May 20, 2025: final version

1 Introduction

It is not uncommon to read that a new version of Android fixes 50 or more security vulnerabilities. This recurring pattern raises pressing questions: Where do these issues come from? Why can't they be prevented? And why do updates often introduce new vulnerabilities of their own? We believe many of these problems originate from fundamental architectural choices made at the lowest levels of modern computing systems.

Most mobile platforms, for example, are built on ARM processors that follow the von Neumann architecture—a design where code and data share the same memory space. While efficient and flexible, this unified memory model is inherently vulnerable to security issues such as buffer overflows and code injection attacks. Compounding the risk are complex software stacks, heavily reliant on reusable libraries and written in languages like C and C++, which offer performance and control but little inherent memory safety.

To understand the security implications of these foundational choices, it is useful to consider a contrasting model. The Microchip PIC16F88, a classic 8-bit microcontroller, offers an instructive counterexample. Based on the Harvard architecture, it physically separates program code from data. This architectural distinction prevents the execution of injected or self-modified code, thereby eliminating entire classes of attacks that plague von Neumann machines. For instance, in a Harvard system, a buffer overflow cannot overwrite executable code—unless special access modes, like low-voltage programming (LVP), are explicitly enabled.

Admittedly, comparing a low-power PIC16F88 to a modern ARM processor may seem incongruous due to vast differences in computational power, memory, and peripheral support. However, our goal is not to equate them but to examine whether lessons from simpler, more constrained environments can guide the design of safer and more predictable computing systems. The PIC16F88 serves as a minimal yet concrete platform for testing these architectural ideas.

Yet the Harvard architecture is not without its trade-offs. The PIC16F88, for instance, employs fixed-length instructions and lacks a true data stack. This limits programming flexibility, particularly in languages that depend on stack-based function calls and local variable storage. Developers must manually simulate stack behavior using static memory, which complicates subroutine design and parameter passing. This challenge is especially evident in structured programming contexts such as the Ceres system, where modular design relies heavily on well-defined calling conventions.

An additional layer of complexity arises from code paging. On the PIC16F88, program memory is divided into discrete blocks, and calls across page boundaries must be handled manually. While this design imposes development constraints, it also fosters a tightly controlled and deterministic execution model—qualities increasingly rare in today's layered and opaque software environments.

These constraints make traditional languages like C and C++ ill-suited for the platform. Both were designed with von Neumann architectures in mind, relying on assumptions such as a unified memory space, implicit stack management, and dynamic allocation. The absence of a data stack on the PIC16F88 renders many standard C constructs awkward or inapplicable. As a result, developers are forced to rework even simple procedures using static memory, increasing both verbosity and the likelihood of bugs.

Moreover, C/C++ compilers assume runtime features such as reentrancy, recursion, and consistent calling conventions—all of which are difficult or inefficient to implement on devices with minimal RAM and

simple instruction sets. Emulating these features on the PIC16F88 adds significant complexity and often undermines the benefits of structured programming, leading to brittle and non-idiomatic code that is hard to maintain.

High-level alternatives like Python also face hurdles in this environment. Python relies on bytecode interpretation or virtual machine execution, which entails considerable runtime overhead. This makes it unsuitable for extremely resource-constrained systems that lack an operating system and have only a few hundred bytes of RAM. Nevertheless, our prior experience with the Python-on-HP65 project demonstrated that a compiled subset of Python could be adapted for such environments. That work proved the feasibility of running high-level constructs on limited hardware and provided valuable insights for future language design.

These insights eventually led us to a pivotal conclusion: rather than retrofitting existing high-level languages, a more sustainable path forward lies in designing a new language from the ground up—one tailored to the constraints and features of Harvard-architecture microcontrollers. We began developing this new language using a modified version of the LabVIEW-based compiler initially created for the HP-65 project. Our objective is to retain the expressiveness and structure of high-level programming while producing efficient and predictable machine code that respects the strict memory and execution model of devices like the PIC16F88.

Firmware development for microcontrollers is intrinsically complex, owing to their close integration of software and hardware. Developers must manage tasks such as sensor data acquisition, actuator control, and communication via UART, I^2C, or SPI with precise timing and minimal overhead. They also contend with digital I/O, analog signal processing, ADCs, PWM, and transient event capture. Interrupt-driven programming and, in some cases, limited multitasking introduce additional challenges, including data corruption, race conditions, and the need for synchronization mechanisms like semaphores or finite state machines.

In such a demanding environment, simplicity and predictability are not luxuries—they are necessities. Our aim is to provide a language and toolchain that embraces these constraints rather than fights them, enabling developers to write safe, maintainable, and efficient code even for the most limited microcontrollers.

1.1 Data Safety Concerns

A major motivation for creating a new programming language tailored to MCU firmware arises from long-standing safety and correctness issues in C and C++ — the dominant languages in embedded systems. Many of these issues have driven the development of safer alternatives, such as Rust. Critical problems in C include:

- Lack of memory safety: C does not perform bounds checking on arrays, leading to buffer overflows. For example, a faulty alarm handler writing past the end of an array could overwrite adjacent memory, potentially corrupting interrupt vectors or control flags.
- No stack overflow protection: Recursive function calls or deep call stacks can silently exceed the available stack space, especially dangerous on small MCUs with limited memory.
- Data races: C lacks built-in concurrency control, making it easy to introduce race conditions in interrupt service routines (ISRs) that access shared data without atomic protection or proper locking mechanisms.
- Use-after-free and dangling pointers: Although less common in bare-metal firmware, misuse of dynamic memory can still occur, particularly in more advanced or reusable embedded frameworks.
- Undefined behavior: Many operations in C result in undefined behavior (e.g., signed integer overflow), which compilers are free to optimize away in unpredictable ways –posing a significant risk in safety-critical firmware logic.

Rust was specifically designed to address these issues through a strict ownership model, lifetimes, and type-safe abstractions that make many classes of bugs impossible to express. However, Rust's steep learning curve and the lack of transparency in code generation can be a barrier in low-level embedded development where precise control over hardware is essential.

This project proposes the design of a lightweight, educational programming language that blends low-level transparency with safer programming constructs. The starting point is the Microchip PIC16F88, a small yet powerful 8-bit MCU built on the Harvard architecture, which inherently separates code and data memory, reducing the attack surface for many types of exploits. The PIC16F88 serves as an ideal testbed due to its simplicity and reliability, while still supporting rich peripheral functions. The recent release of the PIC16F13145

— which adds configurable logic blocks and peripheral pin selection — reinforces the need for languages that enable safe and comprehensible access to increasingly sophisticated hardware features, without inheriting the pitfalls of C.

Illustrative Example: Type Safety in Expressions (Revised for Ceres) In embedded C programming, implicit type promotion is a frequent source of subtle and critical bugs — particularly when working with limited 8-bit or 16-bit architectures. Consider the following C example:

```
uint16_t sensor_val = 62000;
int16_t offset = -1000;
int16_t result = sensor_val + offset;
```

On many platforms, this code compiles without warnings, even though:
The addition mixes signed and unsigned types.
The result of `sensor_val + offset` may overflow or underflow the bounds of a signed 16-bit integer, leading to wraparound behavior that is undefined or platform-specific.
Worse, if `sensor_val` is promoted to int before the addition (depending on compiler settings and integer promotion rules), the result might temporarily exist in a 32-bit register, hiding overflow during development but failing on the actual target hardware.

How Ceres Handles This Ceres avoids these problems through strict single-type expression rules. In Ceres, the above operation would not compile because it attempts to add a raw16 and an int16 directly. Instead, the programmer must explicitly state where and how the type conversion occurs:

```
raw16 sensor_val:=62000
int16 offset:=-1000, result
result := int16(sensor_val) + offset
```

Why This Matters in Embedded Contexts On small 8-bit MCUs like the PIC16F88, errors due to silent promotions or mixed-type arithmetic can easily go unnoticed during simulation, but lead to serious malfunctions in real-world conditions — such as premature triggering of alarms, incorrect sensor readings, or unstable actuator signals.

By enforcing a strong and explicit type discipline, Ceres eliminates a whole class of runtime errors that are common in C, supports better static analysis, and fosters a deeper understanding of low-level data flow — especially valuable in educational and safety-critical development environments.

Alarm Functions and Overflow Handling in Ceres Additionnally, Ceres uses data safety measures in order to detect and handle overflow conditions during runtime. For instance, this C-code:

```
uint16_t timeout = 65000;
timeout += 1000; // wraps around to 464
```

...will silently wrap and potentially cause a missed timeout or unsafe behavior.

Ceres Approach: Implicite Overflow Mode with Alarms Ceres introduces an implicite alarm-triggered overflow detection.

```
raw16 timeout:= 65000
timeout:=timeout+1000
```

This will trigger a run-time overflow alarm, because the addition ended in a Carry condition. The alarm could be handled by a system-wide trap, a specific error handler, or a controlled shutdown procedure. This could tie into interrupt service routines or watchdog behavior — essential for MCUs like the PIC16F88, where safe and predictable operation is a must. This model gives developers full awareness and control over overflow behavior — a key departure from C, where such issues often go unnoticed until after deployment.

1.2 Improving Code Readability and Syntax Clarity

Another core motivation behind Ceres is to address the syntactic obscurity and historical inconsistencies of C and C++. While these languages were pioneering in their time, much of their syntax remains rooted in early compiler limitations and backward compatibility.

Examples of cryptic or confusing constructs in C/C++ include:

- The use of semicolons to terminate every statement — an error-prone practice, especially when mixing declaration and control flow.
- The need to declare void in a parameter list to indicate "no parameters", e.g. int foo(void), which is conceptually contradictory.
- Function pointer syntax, which becomes nearly unreadable in real-world applications.
- Cryptic type declarations with prefix modifiers:
  `unsigned long long int* volatile*`
- Inconsistent use of symbols $(*, \&, ->)$ with varying meanings depending on context.

These syntax choices, though familiar to seasoned developers, often resemble "charabia" (gibberish) to newcomers or even experienced programmers trying to audit unfamiliar code. This obscurity hinders education, collaboration, and error detection.

Languages like Python have shown the value of clean and readable syntax, but its reliance on indentation for control structures introduces a new category of bugs. Meanwhile, Pascal-style begin / end blocks are verbose and fall out of favor in modern syntax design.

Ceres takes a hybrid approach: it retains C-style braces ({}) to indicate scope — which are visually clear and familiar — while adopting a minimal and explicit syntax more aligned with modern readability standards.

Example: Cryptic C/C++ vs. Clear Ceres Syntax C/C++ Version

```c
void (*handler)(int) = NULL;

void init(void) {
    if ((status = check_state()) == 1)
        do_something(); else do_something_else();
}
```

This code raises several readability issues:

- The function pointer declaration is hard to parse.
- `void` must be explicitly stated in the parameter list.
- Assignment inside an if condition (using = instead of ==) is legal and a common source of bugs.
- Ternary-like one-line if/else reduces clarity and is hard to debug.

Ceres Version

```
$function pointer declaration is not implemented$
subroutine init() {
    boolean status
    if  ( status = check_state() )  {
        do_something()
    }
    default {
        do_something_else()
        }
}
```

Simplifications in Ceres:

- The `subroutine` keyword is clear. In fact, comparably to PASCAL, Ceres distinguishes parametric subroutines from functions. Subroutines may not be used within expressions, as they do not return any result values.
- No need for void to indicate no parameters — () suffices.
- No implicit semicolon noise.
- Assignment inside conditions is not allowed; comparisons must be explicit.

These design decisions make Ceres easier to read, audit, teach, and maintain, particularly in embedded systems where code clarity can directly impact system safety and developer confidence.

1.3 Simplifying Control Flow with Readable and Predictable Constructs

One of the most problematic aspects of C/C++ in both education and embedded firmware development is its control flow complexity. Legacy features like switch/case blocks with mandatory break statements, deeply nested for loops with multiple control variables, and ambiguous loop entry/exit conditions contribute to bugs, especially when timing and correctness are critical.

1.3.1 Case structures and the "fall-through" hazard

C's switch/case allows fall-through unless a break is added manually — a source of many subtle bugs:

```
switch (mode) {
    case 0:
        set_idle();
    case 1:
        start_motor();
        break;
    default:
        error();
}
```

Here, forgetting a break after case 0 leads to unintended execution of `start_motor()` — a typical and dangerous pitfall in embedded control code.

Ceres Solution: if / altif / default Ceres replaces switch/case entirely with a clean conditional structure using if, altif, and default, comparably to Python. This avoids hidden behavior and makes logic explicit:

```
if mode == 0 { set_idle() }
altif mode == 1 { start_motor() }
default { error() }
```

This structure is easier to read and maintain, safer, with no accidental fall-through, more intuitive for students and developers alike.

1.3.2 Unified Looping with quit

C provides two primary loop constructs: while and do...while, along with additional control statements such as break and continue. For creating an infinite loop, programmers typically use the unconventional while(1) construct.

Ceres introduces a single, powerful loop structure:

```
loop {
    if condition {
        quit    $ exits this loop level $
    }

    $ do something $

    if done {
```

```
            quit
        }
}
```

If a quit appears immediately at the top, the loop behaves like a while. If quit appears at the end, it's like a do...while. A quit anywhere else exits only the current loop level, never affecting if blocks or other control structures. This removes ambiguity and ensures loop exit logic is always intentional.

1.3.3 Simplified and Safe `for` Loops: `take` Syntax

C's for loop is infamous for its flexibility and corresponding complexity:

```
for (int i = 0; i < N; i++) {
    // danger of signed overflow, or incorrect bounds
}
```

In embedded systems, this can lead to issues with signed integers wrapping around, or confusion over inclusive vs. exclusive bounds.

Ceres uses a precise and readable syntax:

```
take i from [0, 10) {
    $ i takes values 0 through 9 $
}
```

Or, inclusive of the upper bound. (Note that this distinction is particularly useful with variable array indices, which typically run from 0 to N-1, where N is the array size):

```
take i from [0, 10] {
    $ i takes values 0 through 10 $
}
```

Key rules: The index variable i must be unsigned for safety. Range boundaries are clear and predictable. Eliminates off-by-one errors and signed overflow risks. Removes complex increment/decrement expressions — all handled by the language.

C/C++ vs. Ceres Control Flow: A Side-by-Side Comparison

```
for (int i = 0; i < 10; i++) {
    if (check_error()) break;
    do_step(i);
}
```

```
take i from [0, 10) {
    if check_error() {
        quit
    }
    do_step(i)
}
```

This simplification strategy in Ceres enhances code clarity, runtime safety, and teaching value. It also reduces the burden on both human readers and static analyzers — making embedded software more robust by design.

2 Hello world

The first compiled Ceres program for the Microchip PIC16F88, assembled using Microchip MPASM and downloaded through Microchip MPLAB, demonstrates two basic yet essential embedded behaviors: it sends the classic "Hello world!" message over UART and simultaneously flashes an LED—both at a steady rate of once per second. As shown in the attached listing (see Listing 1), the program is structured into three major parts. The first part, Configuration, is mandatory and includes selecting the target microcontroller (PIC16F88), setting the oscillator frequency and type, enabling required peripherals (here, UART), and configuring system-level settings such as interrupts and I/O direction. (Note that by default all pins are outputs). The second part, Setup, although optional, is conventionally used and defined in Ceres as a job. It allows for global definitions such as variables, non-parametric subroutines, and snippets—compact inline segments combining intermediate Ceres constructs and raw assembler code. Global elements are accessed using the @ symbol, except for 16F88 system registers (called Special File Registers in Microchip terminology), which are accessible directly from anywhere without the @ prefix.

Snippets adapt to their context: when inserted into a job, function, or subroutine, any variable names inside are locally resolved. For example, a snippet using current will bind to the current variable defined in the surrounding job. To explicitly refer to a global variable in a snippet, the keyword GLOBAL (with a space after) must be used instead of @. Additionally, the keyword LOCAL is permitted in GOTO and LABEL statements within snippets, indicating that the jump is local to the specific context in which the snippet is inserted—necessary when snippets are reused multiple times in the same job or function.

Assembler code embedded in snippets must begin with the cryptic marker %7_, while assembler-defined labels must begin with %0_. Furthermore, a dot (.) preceding intermediary Ceres code signifies that a code segment must be inserted at that precise location; correct parameters must be provided, or compilation will fail. These conventions will be formally explained in later documentation. The third part, the main job, is optional but provides a structural separation between setup and user logic. While all functionality could reside in the setup job, this distinction prepares for future versions of Ceres that will allow user-defined setup extensions. However, in the current prototype, separate saving and loading of setup files is not yet implemented. This release serves as a foundational prototype, emphasizing extensibility, clarity, and a low-level yet structured approach to PIC programming.

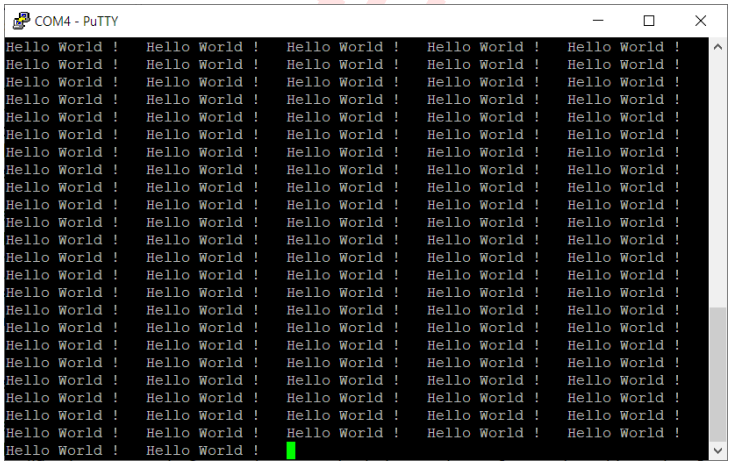

Figure 1: The PIC16F88 MCU sends its messages.

Listing 1: Ceres' program *hello_world.cer*

```
configuration {
   target:=16F88
   frequency:=INTERNAL
   memoryclear:=TRUE
   interrupts:=FALSE
   devices {
      uart {
         baudrate:=9600
```

```
9           parity:=NONE
10        }
11     }
12  }
13
14  setup{
15     array [16] raw8 text
          :=[72,101,108,108,111,32,87,111,114,108,
16           100,32,33,32,32,32]
17
18     snippet delay1000 { .MICRODELAY 1000000 }
19     snippet LED_ON { .SETBIT_raw8 PORTB RB0 0 }
20     snippet LED_OFF { .CLEARBIT_raw8 PORTB  RB0 0 }
21
22     snippet uart_tx_byte{
23           %§7_BANK0
24           LABEL LABEL_WAIT_TX_FREE LOCAL
25           %§7_BTFSS PIR1,TXIF
26           GOTO LABEL_WAIT_TX_FREE LOCAL
27           PUSHV_raw8 current
28           POPV_raw8 TXREG
29     }
30     delay1000
31  }
32
33  main{
34     raw16 index
35     raw8 current
36     loop {
37       LED_ON
38       $delay1000$
39       take index from [0,16) {
40          current:=@text[index]
41          uart_tx_byte
42       }
43       LED_OFF
44       delay1000
45     }
46  }
```

Ceres compiles this program into intermediate code (cf. Listing 2), where snippets are processed and added only in the subsequent recursive process. This recursive approach includes an important code optimization that combines successive and stacked push/pop statements into faster MOV instructions, thereby improving execution efficiency. The intermediate code is then converted into native code via .functions, which are stored in a basic file that contains all the parameterized native code snippets. Notably, these .functions can also contain intermediate code functions. However, it's crucial to note that due to the recursive nature of the process, .SOME_NATIVE may not contain the corresponding SOME_NATIVE function itself, highlighting an important aspect of how recursion impacts the organization and referencing of native functions in Ceres. The configuration part is added at the end of the compiling process. The resulting Assembler programm is shown in Listing 3.

Listing 2: Ceres' intermediate code *hello_world.crA*

```
1  LABEL LABEL_GLOBAL_setup
2  INIT_raw16 GLOBAL sp=239
3  INIT_raw8 GLOBAL text
      =[72,101,108,108,111,32,87,111,114,108,
4              100,32,33,32,32,32]
5  DEF_SNIPPET delay1000
```

```
DEF_SNIPPET LED_ON
DEF_SNIPPET LED_OFF
DEF_SNIPPET uart_tx_byte
SNIPPET delay1000

$here the program implicitly jumps to main job$

LABEL LABEL_main
INIT_raw16 sp=191
INIT_raw16 index
INIT_raw8 current
LABEL LABEL_1001
SNIPPET LED_ON
PUSHC_raw16 0
POPV_raw16 index
LABEL LABEL_1003
PUSHV_raw16 index
PUSHC_raw16 16
LT_raw16
SKIP_T
GOTO LABEL_1004
PUSHV_raw16 index
SETDISPL_raw16
PUSHA_raw8 GLOBAL text
POPV_raw8 current
SNIPPET uart_tx_byte
PUSHV_raw16 index
INCR_raw16
POPV_raw16 index
GOTO LABEL_1003
LABEL LABEL_1004
SNIPPET LED_OFF
SNIPPET delay1000
GOTO LABEL_1001
LABEL LABEL_1002
```

Listing 3: Ceres generated Assembler code *hello_world.asm*

```
;
       LIST    p=16F88

#include "P16F88.INC" ; Include header file

       __CONFIG _CONFIG1, _LVP_OFF&_PWRTE_ON&_INTRC_IO&_MCLR_ON&_WDT_OFF

BANK0    MACRO
         BCF  STATUS,RP0
         BCF  STATUS,RP1
         ENDM
BANK1    MACRO
         BSF  STATUS,RP0
         BCF  STATUS,RP1
         ENDM
BANK2    MACRO
         BCF  STATUS,RP0
         BSF  STATUS,RP1
         ENDM
BANK3    MACRO
         BSF  STATUS,RP0
         BSF  STATUS,RP1
         ENDM

sys_test_j0 EQU 0x20
```

```
sys_X_j0 EQU 0x21
sys_Y_j0 EQU 0x23
sys_Z_j0 EQU 0x25
sp_j0 EQU 0x27
PCLATH_TEMP_j0 EQU 0x29
sys_displ_j0 EQU 0x2A
sys_tmp_j0 EQU 0x2C
W_TEMP_j0 EQU 0x2D
STATUS_TEMP_j0 EQU 0x2E
FSR_TEMP_j0 EQU 0x2F
text_j0 EQU 0x40
sys_test_j2 EQU 0x30
sys_X_j2 EQU 0x31
sys_Y_j2 EQU 0x33
sys_Z_j2 EQU 0x35
sp_j2 EQU 0x37
PCLATH_TEMP_j2 EQU 0x39
sys_displ_j2 EQU 0x3A
sys_tmp_j2 EQU 0x3C
W_TEMP_j2 EQU 0x3D
STATUS_TEMP_j2 EQU 0x3E
FSR_TEMP_j2 EQU 0x3F
index_j2 EQU 0x50
current_j2 EQU 0x52


        GOTO START
        ORG H'04'


START:
        ;initialize Port A
        BANK0
        CLRF    PORTA
        BANK1
        CLRF ANSEL
        MOVLW   0x07
        MOVWF   CMCON
        MOVLW   0x0
        MOVWF   TRISA
        ;initialize Port B
        BANK0
        CLRF    PORTB
        BANK1
        MOVLW   0x0
        MOVWF   TRISB

        BANK1
        MOVLW 0x70
        IORWF OSCCON,F
        ;configure UART
        BANK1
        BSF TRISB,2
        BSF TRISB,5
        MOVLW 0x33
        MOVWF SPBRG
        BSF TXSTA,BRGH
        BCF TXSTA,SYNC
        BANK0
        BSF RCSTA,SPEN
        BCF RCSTA,RX9
        BSF RCSTA,CREN
        BANK1
        BCF TXSTA,TX9
        BSF TXSTA,TXEN



LABEL_GLOBAL_setup

        MOVLW 0xEF
        BANK0
        MOVWF sp_j0
```

```
100         MOVLW 0x0
101         MOVWF sp_j0+1
102 
103         MOVLW 0x48
104         MOVWF text_j0
105         MOVLW 0x65
106         MOVWF text_j0+0x1
107         MOVLW 0x6C
108         MOVWF text_j0+0x2
109         MOVLW 0x6C
110         MOVWF text_j0+0x3
111         MOVLW 0x6F
112         MOVWF text_j0+0x4
113         MOVLW 0x20
114         MOVWF text_j0+0x5
115         MOVLW 0x57
116         MOVWF text_j0+0x6
117         MOVLW 0x6F
118         MOVWF text_j0+0x7
119         MOVLW 0x72
120         MOVWF text_j0+0x8
121         MOVLW 0x6C
122         MOVWF text_j0+0x9
123         MOVLW 0x64
124         MOVWF text_j0+0xA
125         MOVLW 0x20
126         MOVWF text_j0+0xB
127         MOVLW 0x21
128         MOVWF text_j0+0xC
129         MOVLW 0x20
130         MOVWF text_j0+0xD
131         MOVLW 0x20
132         MOVWF text_j0+0xE
133         MOVLW 0x20
134         MOVWF text_j0+0xF
135         MOVLW 0xD0
136         MOVWF sys_X_j0
137 LABEL_1005
138         MOVLW 0x13
139         MOVWF sys_Y_j0
140 LABEL_1006
141         MOVLW 0xA7
142         MOVWF sys_Z_j0
143 LABEL_1007
144         DECFSZ sys_Z_j0,F
145         GOTO LABEL_1007
146         DECFSZ sys_Y_j0,F
147         GOTO LABEL_1006
148         DECFSZ sys_X_j0,F
149         GOTO LABEL_1005
150         NOP
151         NOP
152         NOP
153         NOP
154         NOP
155 
156         BCF PCLATH,4
157         BSF PCLATH,3
158         GOTO LABEL_main
159 
160         ORG 0x800
161 
162 LABEL_main
163 
164         MOVLW 0xBF
165         BANK0
166         MOVWF sp_j2
167         MOVLW 0x0
168         MOVWF sp_j2+1
169 LABEL_1001
170         BANK0
171         BSF PORTB,RB0
172 
```

```
        MOVLW 0x0
        MOVWF index_j2
        MOVLW 0x0
        MOVWF index_j2+1
LABEL_1003

        MOVLW 0x10
        BANK0
        MOVWF sys_Y_j2
        MOVLW 0x0
        MOVWF sys_Y_j2+1

        MOVF index_j2,W
        MOVWF sys_X_j2
        MOVF index_j2+1,W
        MOVWF sys_X_j2+1
        MOVF sys_Y_j2,W
        SUBWF sys_X_j2,F
        MOVF sys_Y_j2+1,W
        BTFSS STATUS,C
        INCFSZ sys_Y_j2+1,W
        SUBWF sys_X_j2+1,F
        MOVLW 0x0
        BTFSS STATUS,C
        MOVLW 0xFF
        MOVWF sys_test_j2
        BTFSS sys_test_j2,0
        GOTO LABEL_1004

        BANK0
        MOVF index_j2,W
        MOVWF sys_displ_j2
        MOVF index_j2+1,W
        MOVWF sys_displ_j2+1
        MOVLW 0x40
        ADDWF sys_displ_j2,F
        BTFSC STATUS,C
        INCF sys_displ_j2+1,F
        MOVLW 0x0
        ADDWF sys_displ_j2+1,F
        BCF STATUS,IRP
        BTFSC sys_displ_j2+1,0
        BSF STATUS,IRP
        MOVF sys_displ_j2,W
        MOVWF FSR
        MOVF INDF,W
        MOVWF sys_Z_j2

        MOVF sys_Z_j2,W
        MOVWF current_j2
LABEL_WAIT_TX_FREE_j2_3
        BTFSS PIR1,TXIF
        GOTO LABEL_WAIT_TX_FREE_j2_3

        BANK0
        MOVF current_j2,W
        MOVWF TXREG

        MOVF index_j2,W
        MOVWF sys_X_j2
        MOVF index_j2+1,W
        MOVWF sys_X_j2+1
        INCF sys_X_j2,F
        BTFSC STATUS,Z
        INCF sys_X_j2+1,F

        MOVF sys_X_j2,W
        MOVWF index_j2
        MOVF sys_X_j2+1,W
        MOVWF index_j2+1
        GOTO LABEL_1003
LABEL_1004
        BANK0
```

```
246         BCF PORTB,RB0
247         MOVLW 0xD0
248         MOVWF sys_X_j2
249 LABEL_1008
250         MOVLW 0x13
251         MOVWF sys_Y_j2
252 LABEL_1009
253         MOVLW 0xA7
254         MOVWF sys_Z_j2
255 LABEL_1010
256         DECFSZ sys_Z_j2,F
257         GOTO LABEL_1010
258         DECFSZ sys_Y_j2,F
259         GOTO LABEL_1009
260         DECFSZ sys_X_j2,F
261         GOTO LABEL_1008
262         NOP
263         NOP
264         NOP
265         NOP
266         NOP
267         GOTO LABEL_1001
268 LABEL_1002
269 LABEL_end_main
270         GOTO LABEL_end_main
271         END
```

Please stay tuned and check this site for the next parts that are coming soon. These will be released step by step, starting with the current documentation as-is, followed by updates and refinements. Soon after, we'll begin sharing insights into the compiler internals. Once the prototype passes its tests, the compiler itself will be made available for download directly from the site. We would be grateful for any comments, suggestions, or questions during this process. And don't forget—our long-term goal is to extend Ceres to support the PIC16F13145 and successfully run this program, along with many more, on it using the Ceres framework.