MCUnity SDD

|  |  |
| --- | --- |
| MCUnity application Software Design Document | 01/09/2016 |

# MAJOR DESIGN ISSUES

## Re-Setup Request

When the app crashes and restarts, it doesn’t know where the MCU’s are. It does its broadcast but no MCU will reply to it any longer (this only happens once). So there’s no way for the application to know which IP addresses to send a Re-Setup Request.

SOLUTION: make that request a broadcast. Unlike the initial broadcast, it should be non-recurring.

I can’t just make the initial broadcast non-recurring : the MCU needs to behave differently when it connects to the app the first time and when a re-setup is requested, especially if there are different MCU’s connecting to the app at different times (after and before the app’s crash / restart)

# IDEAS TO TURN INTO SPECS

## Calling functions on the MCU

Functions can be stored as an array of pointers and therefore handled like any other variable by MCUnity. However this will have to be limited to functions without arguments nor return values.

## GUI Layout Setup

The order in which the ESP registers GUI elements will obviously matter. But to make the most out of high-resolution displays I need a way to specify “carriage returns” and the width (in percentage?) of each GUI element.

Or I can go for a tile-based approach.

## Support for Multiple Simultaneous ESP’s

The idea is to also associate their IP address to each tile.

In terms of layout, things can get complicated. Perhaps a tabbed screen is the best approach?

I could also imagine a system where the user can rearrange tiles to get the combination of “views” he needs.

# Abstract / Project Summary

MCUnity is a firmware development tool. It’s essentially the combination of a Unity application and a library for an MCU that provides a nice user interface with the following capabilities:

* Accessing global variables and ESP registers
* Launching tasks, calling functions
* Displaying text from ESP to Unity

The Unity application is generic, i.e. not tailor-made for a specific microcontroller family or firmware project. Its GUI is configured at run time and from the firmware. This keeps the user focused on MCU firmware development.

## Modular Design

Designing and programming MCUnity requires working simultaneously on two different fronts:

* Unity development in C#
* MCU development in C/C++

This requires two different sets of tools.

To help deal with the complexity, MCUnity is designed to be very modular. This document both describes how the existing features of MCUnity are implemented and how to add new ones.

This modularity also makes it easier to propose **frequent updates**.

# System Architecture

An MCUnity system is a set of:

* One computer running the MCUnity application
* One or more microcontrollers (MCU) running firmware using the MCUnity library

MCU do not communicate with each other as part of the MCUnity system, although they may communicate with each other for unrelated purposes. Therefore, the MCUnity system has a **star topology**. To keep this document simple, the author assumes there’s only one MCU in the system, unless otherwise specified.

## Unity Application

The Unity application is intended to be completely generic, meaning it should not need to be modified to suit a particular type of MCU or a particular MCU firmware project. The application should be seen as a monolithic tool to be controlled from an MCU.

This requirement has crucial aims:

* Limit MCUnity system development to MCU programming only.
* Remove the need for installing / reinstalling multiple variants of the Unity application.

## MCU Library

MCUnity is a software debugging tool, although it may also be used to provide a remote GUI to an MCU. In either case, it is required for the MCU firmware to use the MCUnity library to perform the following tasks:

* Establish connection with the Unity application
* Setup the Unity application’s GUI
* Respond to GUI commands
* Update the GUI contents

The MCUnity library is designed so that it can be built optionally into a firmware project. This means it can be excluded from release builds. Given its size and RAM utilization, but also the fact that it is technically a network backdoor, it is recommended to exclude the MCUnity library from any firmware release. **Because of its polymorphism and intended purpose, MCUnity is neither an optimized GUI system nor a secure GUI system**.

### Target Hardware and System Requirements

MCUnity is primarily designed for 32-bit microcontrollers.

MCUnity requires a local network connection to the machine that runs the Unity application. That can be either a WiFi or Ethernet connection.

MCUnity relies on the lwIP network stack.

CPU performance and memory requirements **TBD**.

# Terminology

Unless otherwise specified:

**Application** describes the MCUnity Unity application. The application is programmed in **C#**.

**Firmware** describes the firmware running on the MCU. The MCUnity library for the firmware is programmed in **C**.

**Opcode** (operation code) describes a unique 8-bit operation identifier.

**Operation** describes an MCUnity operation where the firmware sends a command to the application or the application sends a command to the firmware.

**Packet** describes the payload of a UDP packet sent between the application and the firmware on the MCUnity port (55555, or “five fives”).

**Payload** is synonymous for **packet**, as defined above.

# Network Protocol

MCUnity uses UDP, with port 55555 (five fives).

In keeping with the choice of UDP, which is connectionless, the MCUnity protocol is based entirely on single packets.

MCUnity UDP transfers are strictly **point to point** except for one situation: mutual discovery. Initially, the application and the firmware don’t know each other’s IP address. The application periodically sends a UDP broadcast packet on the MCUnity port which will be received by any MCU on the LAN that is running the MCUnity library. This will let the MCU know the IP address of the machine that runs the application. From then on, what logically follows is a phase where the MCU firmware will setup at least one GUI element, requiring transmission of one UDP packet to the application. Upon receiving that packet, the application will know the IP address of the MCU.

For the purposes of this document, the terms “packet” and “payload” are interchangeable and both refer to the payload of a UDP packet sent between the application and the firmware on the MCUnity port.

All packets start with a single-byte opcode. The format of the rest of the packet is inherent to the opcode.

A packet may contain at most one variable-length field, which must always be the last field in the packet. This is to eliminate the need to generate, transmit and decode field length fields.

In a general fashion, variable-length packets are not recommended. Keep in mind that MCU have very limited processing and memory resources. Always attempt to use a fixed-length packet format when implementing new operations. Consider that it will be faster for the firmware to call a fixed-length operation a variable number of times than it would be to generate or decode variable-length packets.

# MCUnity Operations

An “operation” is a transaction between the MCU and the Unity application. It takes the form of a function call on the transmitting side, resulting in transmission of a single UDP packet, which triggers a specific response on the receiving side.

Other Side  
(Firmware or Application)

Operation Initiator  
(Application or Firmware)

One UDP packet

An operation is initiated by calling a dedicated **function** (in C) or **method** (in C#).

The application uses a thread to process incoming UDP packets. This thread parses the packet to determine the operation code (or opcode) and then passes on the packet to the relevant method.

The firmware uses an lwIP callback to process incoming UDP packets. Ideally, this callback should not call other functions.

Therefore, implementing an MCUnity operation takes the following steps:

* Protocol extension:
  + Assign a unique 8-bit operation code, or opcode, to the operation
  + Define its UDP packet’s payload format
* Implementation of the function (firmware) or method (application).
* Implementation of the parser method (application) or callback code block (firmware).

The protocol is described both in this document and in a spreadsheet stored with this document.

This section documents each supported operation ranked by rising opcode. In the following subsection titles those are indicated in the form “Ixxx” where “I” denotes the initiator (A for Application, F for Firmware) and “xxx” is an integer in the range 1 to 255.

## (A001) Application Broadcast

Purpose: periodic broadcast intended to let the firmware discover the IP address of the machine running the application.

Note: the broadcast is set to repeat at 1 Hz, which is considered a good compromise between network load and system responsiveness. Unlike all other packets sent by the application, the broadcast is send from a dedicated thread: future versions of the application may implement a feature to turn it off or activate it on user command.

Upon receiving this packet, the firmware saves its source IP address and transitions from mode “init” to mode “setup”. If the broadcast packet is received while the firmware is not in mode “init”, no state transition occurs.

Firmware variables that are modified:

* unity\_IP IP address of the machine running the MCUnity application
* unity\_mode MCUnity system state (firmware copy)

## (A002) Force Setup

Purpose: in case the application exits and restarts while the firmware is still running, causes the firmware to perform its setup operations again.

This also causes the firmware to transition from “update” mode back to “setup” mode while this operation is performed. This feature may be useful if the sequence of setup operations can be interrupted by FreeRTOS and another part of the firmware attempts to update the remote GUI.

## (x003) <reserved>

## (F004) Setup “int” Type Variable

Setup functions are used by the firmware to add controls to the GUI generated by the application.

This specific operation provides the application with the parameters for setting up a GUI control for an “int” variable (signed 32-bit integer):

* The variable’s unique index, which allows the application and the firmware to refer to the same variable without using its name or address.
* The variable’s current value, which the application will use to initialize the control.
* Minimum and maximum values that serve two distinct purposes:
  + They bind user input to a range that is compatible with the firmware.
  + They can help define the appearance of the GUI control or its operation, for example if the variable is set to be displayed as a percentage or a progress bar.
* A 32-bit “options” field to define the appearance of the GUI control and/or its modes of operation.
* The variable’s “display name”, a character string the application will use to label the control. It doesn’t need to match the variable’s name and can contain any ASCII character including spaces.

The unique index is generated by the MCUnity library, which also reads the “int” variable’s current value. The remaining fields are passed as arguments to the MCUnity firmware function that initiates this operation:

int unity\_setup\_int (int\* variable, const char\* name, int min, int max, uint32\_t flags);

Implementation note: each firmware variable type has a dedicated trio of operations:

* “Setup” on the firmware side, registers a variable with the MCUnit library and sends a command to the application to create the matching GUI control.
* “Set” on the application side, sends a new value to be assigned to a variable.
* “Update” on the firmware side, sends the value of one or several or all variables of the type to the application, so it can update the GUI.

### Application Side

Upon receiving an F002 packet…

## (A005) Set “int” Type Variable

“Set” operations are used by the application to set the value of a variable in the firmware. In this case the variable to be set is of type “int” (32-bit signed). The packet contains the opcode (0x04), the unique index of the variable, and its value on 32 bits.

Implementation note: due to the fact that each variable type has its own “Setup”, “Set” and “Update” operations, the unique index of a variable only needs to be unique among variables of the same type.

## (F006) Update “int” Type Variables

Purpose: sends the current value of all registered “int” type variables to the application to update the GUI.

Note: this approach is preferred over updating single variables as it reduces network load. The same doesn’t apply in the other direction (A003) because it is assumed a human user of the GUI cannot update more than one variable at a time, or within the time it takes the application to build and transmit a packet.

## (A007) Request “int” Update

# Packet Field Formats

## GUI Control Options Field

This is a 32-bit field present in:

* All “setup” operation packets

It controls the appearance and modes of operation of a GUI control.

THE FOLLOWING IS TENTATIVE AND NOT YET IMPLEMENTED

Sub-fields:

* GUI layout:
  + Control width: **2 bits** to set a control’s width to 25%, 50%, 75% or 100% of the overall width of the GUI.
* GUI control type: this field dictates which type of Unity GUI control will be used. Control type dictates whether the control is interactive or not (i.e. text field versus input field).
* GUI control’s color 1: 3 bits to set the color of the variable’s value
* GUI control’s color 2: 3 bits to set the color of the control itself

# Firmware global variables

# Application data members (or fields ? see C# doc)

# Support for Multiple MCU

PRELIMINARY: USE TABS

*Also, simply add a dimension to all arrays used to store GUI setup information, that index identifying the MCU. MCU indices can be derived from the last byte of their IP address (on a LAN) going through a LUT to keep the arrays at a reasonable size on mobile devices.*

# Firmware Design with MCUnity

This may need to go into some sort of user / developer guide.

Here are the rules and best practices for integrating MCUnity into a firmware design.

## Location

In order to keep firmware source code clean and make it easy to exclude MCUnity calls from compilation, MCUnity should be integrated only in the firmware’s main source file.

## Include the MCUnity Header File

Add this statement to your firmware’s main source file:

#include “unity.h”

## Declare Global Variables

MCUnity is designed to allow access to a firmware’s global variables from a remote Unity application through a local network. It therefore requires global variables.

If possible, make the firmware’s “variables of interest” global.

## Write a GUI Setup Function

This function must take no argument and have no return value, i.e. its prototype should be in the form:

void gui\_setup\_function (void);

This function must contain the complete sequence of the firmware’s calls to individual MCUnity setup functions, and nothing else. This call sequence can be seen as the specification of the GUI that the MCUnity application will generate and maintain for the firmware.

A pointer to this function will be passed to MCUnity using the “unity\_init” function. This is to allow the remote Unity application to request the firmware execute its MCUnity setup sequence again. That is useful if the application is restarted while the firmware is running.

Centralizing GUI initialization also help development by making it easy to exclude MCUnity calls from compilation. It also makes it easy to maintain several different GUI setup sequences, to be used for investigating different aspects of the firmware’s operation.

This function must **never** be called directly by the firmware.

## MCUnity Initialization

MCUnity should be initialized as soon as possible **after** the firmware establishes a connection to the local network and acquires an IP address.

This is done by calling the “unity\_init” function (no return value).

“unity\_init” takes one argument: a pointer to the GUI setup function (see previous section).

It also sets up the network socket that will be used by MCUnity to communicate with the remote application.

## Wait for MCUnity to Connect

The MCUnity application will periodically (1 Hz) send a broadcast packet to let the firmware learn its IP address. Until the firmware receives and processes this broadcast message, all other MCUnity functions will not work.

To determine the current state of the connection between the application and the firmware, use the dedicated “unity\_not\_ready” function (no argument) which returns 1 until the connection is established, then 0.

The exact implementation of the wait loop is firmware-specific. For FreeRTOS-based firmware it is best to do it inside a task and use “vTaskDelay” to let other tasks execute until the connection is established (assuming that makes sense).

## Remote GUI Setup

Once “unity\_not\_ready” returns zero, it means MCUnity has transitioned out of the “init” state. It’s now possible to send GUI setup operations to the application. Those have been centralized in a firmware-specific function that has been passed to MCUnity. This function must be called exclusively through MCUnity, so it “knows it happened”.

Do this by calling “unity\_setup” (no argument, no return value). Among other things, this function will call the firmware’s custom GUI setup function.

Once this function returns, the MCUnity application is ready to interact with the MCUnity library and the firmware. MCUnity is in “update” mode.

## Update the Remote GUI

TO DO: INDICATE HOW TO PERFORM A REMOTE GUI UPDATE

Updates go both ways, but use two very different processes:

User input on the Unity application is sent to the firmware which makes it effective. This takes the form of update operation packets which are processed asynchronously (as received) by MCUnity’s UDP packet parser.

However, the MCUnity library is not capable of monitoring each variable registered with the remote GUI to send back update operations when one changes. This would require too much processing power.

Instead, MCUnity relies on the **firmware developer** to explicitly trigger remote GUI updates.

The exact placement of these update operations in a firmware depend entirely on that firmware, of course, but optimal placement follows simple, logical rules:

* If the GUI is never updated, it’s little more than an input device. This could be perfectly fine in some projects, for example if the impact of changing a variable from the GUI is observable through other means, like an actuator’s motion, an LED or a display driven by the MCU.
* If the GUI is updated too often, it steals processing power from the firmware’s true function. It may also generate too much network traffic. There’s no point updating a GUI:
  + More often than the refresh rate of the display that GUI is rendered on.
  + When the variables displayed by the GUI don’t change.

Thus, optimal placement is a trade-off. The goal should be to provide the human user with a reasonably responsive GUI without penalizing firmware performance.

A simple, good approach can be to trigger a GUI update from a FreeRTOS task so that it executes at a fixed frame-rate. This can be improved by adding a simple flag (a global variable) the firmware can set to indicate that a GUI update is necessary. This flag can be tested by the GUI update task and then cleared.

# Design Justification

## Centralization of setup operations

There are two possible situations that a firmware designer may want to account for:

* The **application** may already be running when the **firmware** starts running.
* The **firmware** may already be running when the **application** starts running.

For example, the device running the application may crash while the firmware is still running and the user may want to resume operation without resetting the MCU. Likewise, the firmware may crash and require the MCU be reset, but this shouldn’t also require exiting and restarting the application.

To help support these cases, the MCUnity library (firmware side) must be able to send the exact same GUI setup operations at any point during the execution of the firmware. This is what creates the need to encapsulate and centralize the setup operations.