UNIVERSITÀ
DEGLI STUDI
DI PALERMO

NDS
NETWORKING AND DISTRIBUTED SYSTEMS
RESEARCH GROUP

Ambiente simbolico per la configurazione hardware e la programmazione di applicazioni IoT

Tesi di Laurea Magistrale in Ingegneria Informatica

Leonardo Giuliana

Relatore: Prof. Daniele Peri
Correlatore: Ing. Gloria Martorella

Ambiente simbolico per la configurazione hardware e la programmazione di applicazioni IoT

Tesi di Laurea di
Dott. Leonardo Giuliana

Relatore:
Prof. Daniele Peri

Controrelatore:
Prof. Giuseppe Lo Re

Correlatore:
Dott. Ing. Gloria Martorella

Sommario

I recenti progressi tecnologici nel contesto dell'*Internet of Things* (IoT) hanno permesso lo sviluppo e la diffusione di piccoli dispositivi a basso consumo e a basso costo capaci di comunicare tra loro attraverso tecnologie wireless, creando reti distribuite costituite da un gran numero di nodi autonomi e intelligenti capaci di interagire tra loro e di raggiungere obiettivi complessi. In questa tesi si descrive lo sviluppo di un ambiente simbolico per configurare e programmare in tempo reale questi dispositivi senza la necessità di dovere utilizzare complessi meccanismi di compilazione ma utilizzando un semplice linguaggio interpretato inspirato al FORTH. L'ambiente simbolico implementato permette allo sviluppatore di avere accesso alle componenti di livello più basso tramite un linguaggio ad alto livello e di modificare in tempo reale il comportamento dei nodi della rete. I vari nodi della rete ricevono stringhe di codice simbolico da remoto senza che l'utente debba effettuare necessariamente l'accesso fisico al dispositivo. All'interno dell'ambiente simbolico si sono realizzati dei meccanismi, basati su sistemi client/server, per mettere i vari dispositivi nelle condizioni di potere scambiare autonomamente queste stringhe per potere eseguire compiti distribuiti implementando così la logica applicativa direttamente sui nodi della rete.

Indice

Introduzione **1**

1 Stato dell'Arte **3**

2 Sistema **6**

2.1 Ambiente (Mondo Fisico) 6

2.2 Hardware 8

2.3 Software 9

3 ESP8266 ..... **14**

3.1 CPU 15

3.1.1 ISA 15

3.2 Memoria 17

3.3 PIN 17

3.3.1 GPIO 18

3.3.2 Serial Peripheral Interface (SPI/HSPI) 19

3.3.3 I2C Interface 19

3.3.4 I2S Interface 19

3.3.5 Universal Asynchronous Receiver Transmitter (UART) 19

3.3.6 Pulse-Width Modulation (PWM) 20

3.3.7 IR Remote Control 20

3.3.8 Analog-to-Digital Converter (ADC) 20

3.4 Wi-Fi 20

4 Ambiente Simbolico **21**

4.1 Stack 21

4.2 Dizionario 22

4.3 Strutture di Controllo 24

4.3.1 Strutture di Controllo Condizionali 24

4.3.2 Strutture Condizionali Iterative 25

4.4 Eccezioni 26

4.5 Quotation e Combinator 26

4.5.1 Quotation 27

4.5.2 Combinator 27

4.6 Create: e Does> 28

4.7 ESP8266 29

4.7.1 Wi-Fi 30

4.7.2 GPIO, ADC e PWM 31

4.7.3 Network 33

4.7.4 Task e Mailbox 38

4.7.5 Flash 41

4.7.6 TCP REPL 44

5 Ambiente di sviluppo **47**

5.1 Scheda di sviluppo 47
5.2 Download e installazione 49
5.3 Tools 50
5.3.1 flash.py 50
5.3.2 esptool.py 51
5.4 Struttura di Punyforth 54
5.5 Compilazione 57

6 Implementazione 59

6.1 Modifiche al server TCP REPL di Punyforth 59
6.2 Interazione con il sistema di regole 61
6.3 Interazione sicura con il sistema di regole 64
6.3.1 Implementazione all'interno dell'ambiente simbolico 66
6.4 Comunicazione tra nodi tramite TCP REPL e TCP Client 71
6.5 Comunicazione tra nodi tramite MQTT 73
6.5.1 Implementazione all'interno dell'ambiente simbolico 74
6.6 Configurazione dei nodi 78

7 Valutazione Sperimentale 81

7.1 Memoria 81
7.2 Dimensione dell'immagine binaria e dei sorgenti 83
7.2.1 Immagine binaria del Sistema Operativo con l'SDK 83
7.2.2 Immagine binaria di Punyforth 84
7.2.3 Moduli scritti in Codice Simbolico 84
7.3 Test di Configurazione 85
7.4 Tempo di trasmissione del codice simbolico 87
7.5 Comunicazione tra nodi tramite TCP REPL e TCP Client 89
7.6 Client MQTT 91
7.6.1 Confronto dimensione dei sorgenti Client MQTT e ESP-RTOS-MQTT 92

8 Conclusioni e possibili sviluppi 94

9 Appendice: Codice Sorgente 96

9.1 Punyforth 96
9.1.1 wifi.forth 96
9.1.2 netcon.forth 96
9.1.3 task.forth 100
9.1.4 mailbox.forth 101
9.1.5 tcp-repl.forth 102
9.1.6 macros.S 103
9.2 Implementazione 104
9.2.1 aes.c 104
9.2.2 aes.h 120
9.2.3 aes.S 121
9.2.4 base64.c 121

9.2.5 base64.h 126
9.2.6 base64.S 127
9.2.7 des.c 127
9.2.8 des.h 143
9.2.9 des.S 144
9.2.10 tcp-client.forth 145
9.2.11 mqtt-client.forth 146
9.2.12 utils.forth 151
9.2.13 main.forth 157
9.3 Test 158
9.3.1 Codice generato per la configurazione 158
9.3.2 Script Python per il calcolare il tempo di configurazione (client.py) 158
9.3.3 TCP Client - Codice simbolico per definire il comportamento di esp2 160
Indice delle tabelle 164
Indice delle figure 165

Introduzione

Kevin Ashton, nel 1999, coniò il termine *Internet of Things* (IoT) [1] riferendosi a degli oggetti capaci di raccogliere diverse informazioni sul mondo fisico per poi pubblicarle sul WEB. Oggi, questi dispositivi, a basso costo, a basso consumo e a risorse limitate, sono in grado di portare a termine compiti complessi attraverso la loro capacità di comunicare e condividere servizi [2], consentendo all'IoT una grandissima diffusione in diversi campi applicativi come ad esempio domotica, robotica, biomedicale, monitoraggio in ambito industriale, telemetria, efficienza energetica, assistenza remota, tutela ambientale, rilevazione eventi avversi, Smart City, settore terziario e molti altri.

Quindi, ci si trova ad avere a disposizione una grande quantità di dispositivi "intelligenti" capaci di interagire con il mondo fisico, che riescono a cooperare tra loro grazie al continuo sviluppo di protocolli e servizi Cloud [3] dove, un'agente intelligente centralizzato, si occupa della logica applicativa. Il fatto che la logica applicativa risieda su un server remoto centralizzato porta ad avere alcune problematiche di latenza e potrebbe rappresentare un limite per il funzionamento dell'intera rete. Nel caso in cui invece si volesse implementare una logica locale, cioè residente all'interno di ogni nodo della rete, in modo da evitare le problematiche sopra citate, si presenta il problema di dover ricompilare e caricare (da remoto o fisicamente) le immagini binarie, generando così un grande trasferimento di dati, che può essere parzialmente limitato da aggiornamenti incrementali [4], [5]. Si può continuare a mantenere l'approccio della logica locale utilizzando un approccio basato sull'interpretazione di un linguaggio ad alto livello per la riprogrammazione dei vari nodi. Secondo questo approccio, si inviano delle stringhe contenenti del codice simbolico ai vari nodi della rete; il codice inviato viene interpretato ed eseguito in tempo reale dai nodi interessati. Questo codice (che potrebbe contenere, ad esempio, un cambio della configurazione, una nuova procedura per il raggiungimento di un particolare obiettivo) si può rendere a disposizione dei nodi di destinazione della rete senza dover necessariamente ricompilare e installare l'immagine binaria.

Con questa tesi, si vogliono descrivere tutti i passi effettuati per l'implementazione di un ambiente simbolico da utilizzare nei contesti applicativi che fanno parte dell'IoT. L'idea che sta alla base del lavoro svolto è quella di utilizzare

un linguaggio di alto livello ed interpretato [6] e renderlo adatto per gli ambienti in cui si ha a che fare con dispositivi a risorse limitate *general purpose*, dislocati in posti non facilmente accessibili e che possono essere utilizzati in modi diversi all'interno dello stesso contesto: il programmatore, in questo modo, può inviare una stringa contenente il codice simbolico della procedura da eseguire (o della nuova configurazione) ad un nodo della rete da remoto (quindi senza necessariamente avere un accesso fisico al nodo) e senza dovere ricompilare ed installare al suo interno l'immagine binaria, stabilendo in tempo reale le nuove azioni che ogni nodo può compiere.

Il lavoro è stato svolto attraverso l'utilizzo di un linguaggio ispirato al FORTH; questo linguaggio procedurale, simbolico e orientato allo stack, è particolarmente utilizzato nei sistemi integrati e nei dispositivi a risorse limitate. La scelta di questo linguaggio, oltre alle motivazioni precedenti, è giustificata dal fatto che consente allo sviluppatore, con poche e semplici istruzioni e attraverso un linguaggio ad alto livello, di potere interagire direttamente con l'hardware del dispositivo su cui è installato. In particolare, è stato utilizzato Punyforth, un linguaggio ispirato al FORTH, compatibile con diverse architetture. Estendendo il dizionario di Punyforth, è stato implementato un formalismo semplice e condiviso tra i vari nodi, che consente allo sviluppatore di eseguire la configurazione e l'esecuzione di azioni all'interno dei nodi. All'interno dell'attività progettuale si sono implementati dei protocolli di comunicazione per dare al programmatore la possibilità di programmare da remoto i nodi della rete e per supportare la cooperazione tra di essi senza la necessità di un sistema Cloud o centralizzato, riducendo in questo modo i problemi legati alla latenza.

Questo progetto è stato sviluppato in collaborazione con il Dott. Antonio Montalto con cui è stato implementato anche un agente intelligente scritto in Prolog, il quale, attraverso un insieme di regole, produce delle stringhe contenenti il codice simbolico e rappresentanti la configurazione dei vari nodi e le azioni che essi possono compiere. Questa parte di lavoro è documentato nella sua tesi di laurea magistrale intitolata "Sistema di regole per la configurazione hardware e la programmazione di applicazioni IoT". Si ringraziano il Prof. Daniele Peri, la Dott. Ing. Gloria Martorella, il Prof. Salvatore Gaglio, il Prof. Giuseppe Lo Re e la Prof.ssa Alessandra De Paola del corso di Ingegneria Informatica dell'Università Degli Studi di Palermo, per il supporto fornito per lo sviluppo di questo progetto.

1 Stato dell'Arte

In questo lavoro di tesi, sono stati implementati dei protocolli simbolici che consentano a dispositivi a risorse limitate di poter essere facilmente integrati nell'ecosistema IoT e interoperabili con nodi già esistenti e ricchi in termini di risorse. In particolare, è stato sviluppato un client TCP e una versione ridotta del protocollo MQTT. Pertanto, in questo capitolo, verrà presentata una breve disamina relativa ai protocolli utilizzati nel contesto dell'IoT a partire dal basso, fino all'alto livello.

Grazie alla sua ampia diffusione, l'IoT oggi gode della disponibilità di diversi protocolli standardizzati (IETF, IEEE, ITU) e non standardizzati che operano su alcuni livelli dello *stack* di rete (vedi Figura 1). Nello specifico i livelli interessati da questi protocolli sono: collegamento, rete e applicazione [7].

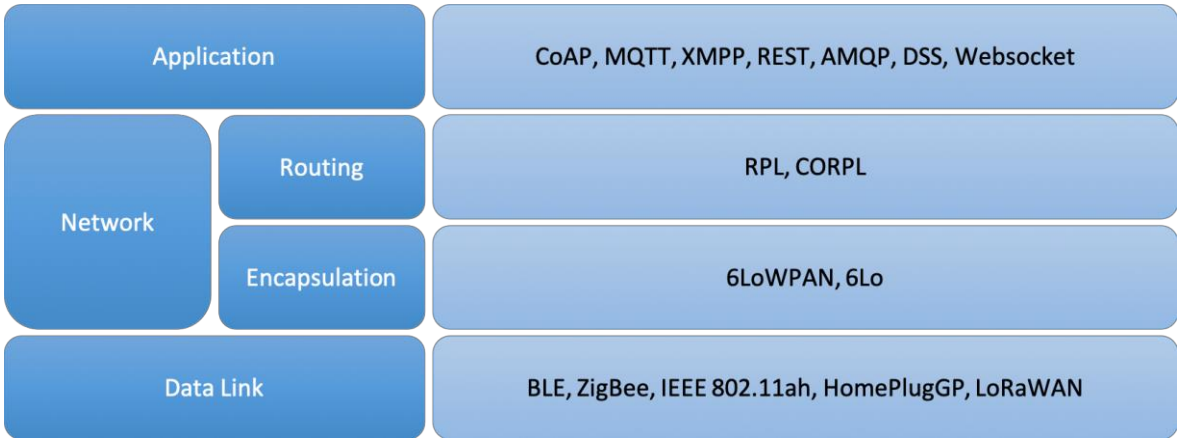

Figura 1 - Stack dei Protocolli IoT

All'interno del *livello di collegamento* troviamo tra i più diffusi il *Bluetooth Low Energy* (BLE) [8], lo *ZigBee* e l'*IEEE 802.11ah* [9]. Quest'ultimo è una versione a basso consumo energetico dell'IEEE 802.11 e ha trovato la sua ampia diffusione grazie alle infrastrutture di rete già esistenti basate su *IEEE 802.11*. All'interno di questo livello troviamo anche altre soluzioni proposte (ritenute più affidabili) come *HomePlug GreenPHY* (HomePlugGP) e *LoRaWAN*.

All'interno del *livello di rete* si trova la distinzione tra protocolli per l'instradamento e protocolli per l'incapsulamento (necessari in quelle applicazioni in cui si usa l'IPv6). Tra i protocolli per l'instradamento il più diffuso è il *Routing Protocol for Low-Power and Lossy Networks* (RPL) [10]. Esiste una versione non standard dell'RPL chiamata *Cognitive RPL* (CORPL) utilizzato nelle reti cognitive

[11]. Tra i protocolli per l'incapsulamento troviamo IPv6 over Low Power Wireless Personal Area Network (6LoWPAN) e 6Lo.

Per il lavoro di tesi svolto, si è voluto dare maggiore attenzione al *livello di applicazione* dove troviamo tutti quei protocolli che gestiscono la comunicazione tra *gateway*, Internet e le applicazioni; il loro compito può essere quello di aggiornare su un server i dati raccolti dai sensori o di comunicare i comandi che gli attuatori devono eseguire. I protocolli utilizzati in questo livello sono:

- *Constrained Application Protocol* (CoAP) [12]: è un protocollo adatto a dispositivi con risorse limitate ed è basato su un sottoinsieme dei metodi HTTP (GET, POST, PUT, DELETE) che usa richieste e risposte sincrone e asincrone. A livello di trasporto usa UDP per ridurre l'utilizzo della banda dovuto all'*overhead* creato da TCP e per potere supportare comunicazioni *multicast*.
- *Message Queue Telemetry Transport* (MQTT) [12]: questo protocollo utilizza TCP a livello di trasporto e si basa sul metodo *publish*/*subscribe* per lo scambio di messaggi, particolarmente adatto per i dispositivi a risorse limitate per il basso uso della banda di rete. All'interno della sua architettura c'è un server (*Broker*) che contiene dei *Topic*. I vari client, chiamati *Publisher* e *Subscriber*, possono rispettivamente pubblicare un messaggio su un determinato *Topic* e ricevere gli aggiornamenti su quel particolare *Topic*. Supporta dei metodi di autenticazione tramite nome utente e password e supporta 4 livelli di *QoS* per i messaggi. Esiste una sua estensione, chiamata *Secure MQTT* (SMQTT), che supporta la meccanismi di cifratura dei messaggi in broadcast.
- *Extensible Messaging and Presence Protocol* (XMPP) [12]: questo protocollo è stato ideato per lo scambio di messaggi all'interno delle chat; solo successivamente è stato utilizzato nel campo dell'IoT. A livello di trasporto usa TCP e per la comunicazione dei messaggi usa i metodi *publish*/*subscribe* e richiesta/risposta.
- *Representational State Transfer* (RESTFUL Services) [12]: il REST non è un vero e proprio protocollo. Usa HTTP e i suoi metodi (GET, POST, PUT, DELETE) per inviare e ricevere messaggi contenenti XML o JSON tramite richieste e risposte sincrone. Usando HTTP, non è usato effettivamente per la comunicazione tra i vari dispositivi ma è usato per le applicazioni finali.

- *Advanced Message Queuing Protocol* (AMQP) [12]: questo protocollo utilizza TCP a livello di trasporto e il metodo *publish*/*subscribe* per la comunicazione di messaggi. Per la sua natura *store-and-forward*, è molto utile nei casi in cui la rete non funzioni correttamente.
- *Data Distribution Service* (DSS) [13]: questo protocollo si basa sul metodo *publish*/*subscribe* per lo scambio dei messaggi, ha 23 livelli di *QoS* e non usa un *Broker* all'interno della sua architettura e non esistono *Topic*: il livello del *Publisher* è responsabile della distribuzione dei dati sensoriali ai *Subscriber*.
- *Websocket* [12]: questo protocollo è stato ideato principalmente in HTML 5 per facilitare i canali di comunicazione su TCP. A partire da un *handshake* eseguito tra client e server, si crea una sessione in cui client e server possono scambiare in maniera asincrona messaggi su una connessione *full-duplex*, riducendo l'*overhead* del polling dell'HTTP. Esiste una versione basata sul metodo *publish*/*subscribe* chiamata *Websocket Application Messaging Protocol* (WAMP). Come il REST, non è usato effettivamente per la comunicazione tra i vari dispositivi ma è usato per le applicazioni finali.

I protocolli esistenti e le loro caratteristiche sono riassunti in Tabella 1.

Protocollo	Trasporto	QoS	Architettura	Dimensione Header (byte)
CoAP	UDP	SI	Request/Response	4
MQTT	TCP	SI	Publish/Subscribe	2
XMPP	TCP	NO	Request/Response Publish/Subscribe	-
REST - HTTP	TCP	NO	Request/Response	-
AMQP	TCP	SI	Publish/Subscribe	8
DSS	TCP/UDP	SI	Publish/Subscribe	-
Websocket	TCP	NO	Client/Server Publish/Subscribe	-

Tabella 1 - Protocolli Livello Applicazione

2 Sistema

All'interno del sistema ideato durante l'attività progettuale sono presenti sia componenti *hardware* che componenti *software*. La componente hardware a cui si rivolge questo sistema riguarda tutti quei dispositivi a basso costo, a basso consumo e a risorse limitate che sono in grado di interagire tra loro, cooperando per il raggiungimento di determinati obiettivi; questi dispositivi fanno parte di una *rete* e ogni dispositivo rappresenta un *nodo della rete*. I vari nodi sono poi inseriti all'interno del *mondo fisico* (*ambiente*) con cui possono interagire secondo le loro capacità. La componente software si occupa di gestire e di mettere in pratica la configurazione di questi nodi garantendo che essi siano in grado di comunicare in modo efficiente secondo un linguaggio condiviso e ad alto livello. Per l'utente e il programmatore, la componente software consente l'interazione immediata con le componenti hardware a basso livello. In particolare, la componente software dell'attività progettuale è costituita da un sistema di regole e da un ambiente simbolico. Lo scambio delle informazioni tra le componenti del sistema è mostrato in Figura 2.

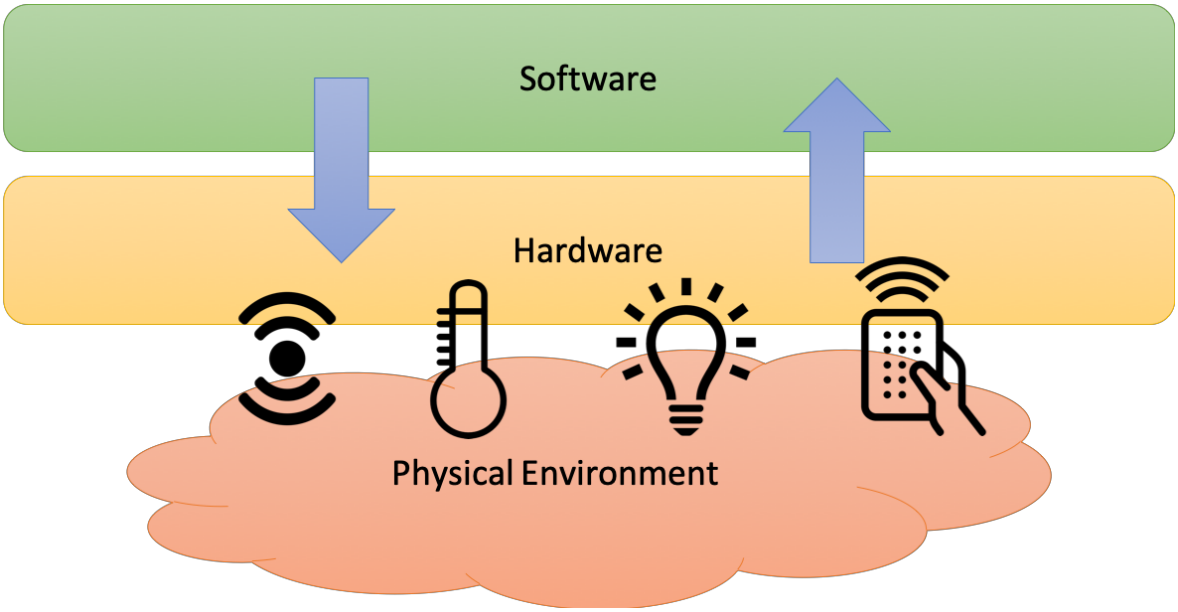

Figura 2 - Scambio di informazioni tra le componenti del Sistema

2.1 Ambiente (Mondo Fisico)

Il mondo fisico è stato rappresentato come costituito da *locazioni*, *oggetti* e *stati*. Una locazione indica una posizione all'interno dell'ambiente (ad esempio: una città, un quartiere, una casa, una stanza, ecc.); in ogni locazione è possibile trovare uno o più oggetti e ogni locazione può avere uno o più stati. Un'oggetto è qualsiasi

cosa con cui il sistema può interagire (ad esempio: una lampada, un sensore, un interruttore, un impianto di condizionamento, ecc.) e con cui è possibile misurare o modificare uno stato. Uno stato è un insieme di parametri fisici che descrivono lo stato dell'ambiente (ad esempio: il "caldo" e il "freddo" sono *stati* definiti dal parametro temperatura con una determinata soglia).

Il mondo fisico assume una struttura gerarchica rappresentata da un albero in cui le locazioni e gli oggetti sono i nodi dell'albero. Ogni locazione può contenere altre locazioni e gli oggetti sono le foglie di questo albero. Tramite questa rappresentazione è possibile ottenere dei percorsi che identificano i vari oggetti e le varie locazioni dell'ambiente (Figura 3).

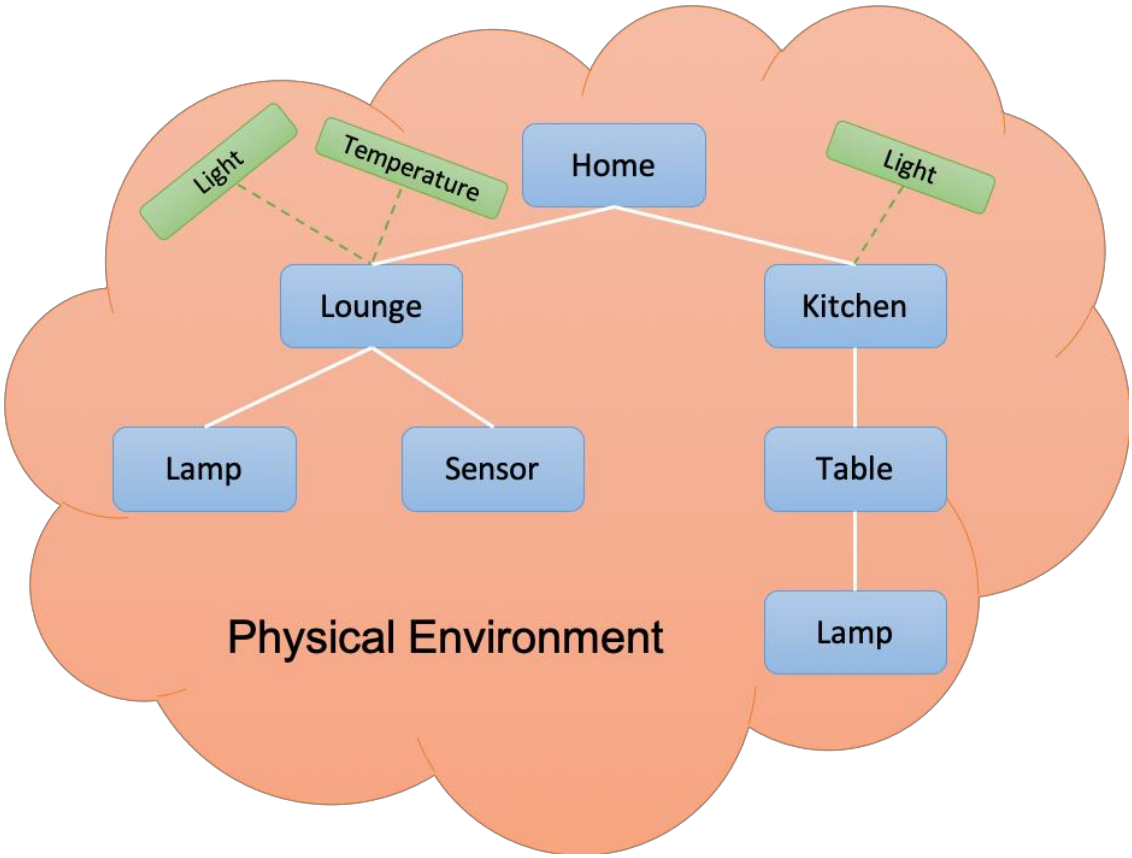

Figura 3 - Struttura ad albero del mondo fisico mediante il quale è possibile identificare oggetti, locazioni e stati

Il sistema tiene conto della struttura dell'ambiente, di ciò che contiene e dei suoi stati per potere effettuare la scelta corretta per raggiungere l'obbiettivo richiesto.

2.2 Hardware

All'interno di questo sistema, l'hardware è tutto ciò che può interagire con l'ambiente e/o raccogliere informazioni sull'ambiente. Al suo interno è possibile descrivere fino al livello di scheda le seguenti componenti: *MicroController Unit* (MCU), *sensori* e *attuatori*. La descrizione a livello di scheda include la specifica formale dei PIN e delle loro funzionalità insieme alle possibili connessioni tra le varie componenti hardware. Ciò presuppone inoltre la descrizione delle caratteristiche di ogni dispositivo e delle interazioni con l'ambiente.

Come MCU, si sono presi in considerazione il *Raspberry Pi* e la *ESP8266* della *Espressif Systems*, nelle due varianti *12-E* e *12-F*. La scelta è ricaduta maggiormente su quest'ultimo MCU non solo per le dimensioni ridotte, il basso costo, l'alta efficienza energetica e la facilità di reperibilità sul mercato, ma anche per la presenza all'interno del SoC di un chip Wi-Fi (IEEE 802.11 b/g/n) con pieno supporto al protocollo TCP/IP grazie ad uno *stack* di istruzioni già implementato all'interno dell'SDK del suo sistema operativo. La presenza di questo chip Wi-Fi e dello *stack* TCP/IP rappresenta un grande vantaggio rispetto ad altri possibili MCU (come *Arduino* e *STM32*) perché in questo modo non c'è la necessità di aggiungere altri moduli hardware e software.

Tra i Raspberry Pi, dal punto di vista degli MCU, si è preso in considerazione il *Raspberry Pi Zero W* perché, come la ESP8266, ha un chip Wi-Fi a bordo e il sistema operativo fornisce uno *stack* completo per il suo utilizzo in rete; rispetto alla ESP8266 però non è economico. Al di là della scelta fatta durante l'attività progettuale, il sistema consente l'aggiunzione di qualsiasi tipo di MCU su cui si può effettuare una descrizione a livello di scheda; l'utente e il programmatore possono utilizzare l'MCU più adatta alle loro esigenze.

Ad ogni MCU possono essere collegati i sensori e gli attuatori in base alla compatibilità dei loro PIN e delle loro funzionalità. Il sistema si riferisce a questi dispositivi chiamandoli *periferiche*. Il sistema, tramite la sua base di conoscenza, è in grado di stabilire tutti i possibili collegamenti in base agli MCU e ai sensori e attuatori definiti; in questo modo il sistema può fornire le possibili configurazioni valide all'utente poco esperto che non conosce esattamente tutte le funzionalità disponibili.

Tra i sensori considerati all'interno dell'attività progettuale, sono stati utilizzati: sensore di corrente (ACS712), di luminosità (GL5516), potenziometro digitale, sensore di Movimento (HC SR501), sensori di gas o di altre sostanze

nell'aria (MQx, con x tra 1 e 12, a seconda delle tipologie di gas che è capace di rilevare), temperatura e umidità (DHT11 e DHT22), temperatura (DS18B20) ed infrarossi. Per quanto riguarda gli attuatori, sono stati utilizzati: relay, trasmettitore IrDA, LED (sia RGB che monocromatici) e buzzer attivi e passivi. Sia sensori che attuatori sono da considerare comprensivi di *shield*, quindi già pronte per il collegamento ai vari MCU. La Figura 4 schematizza l'hardware adottato.

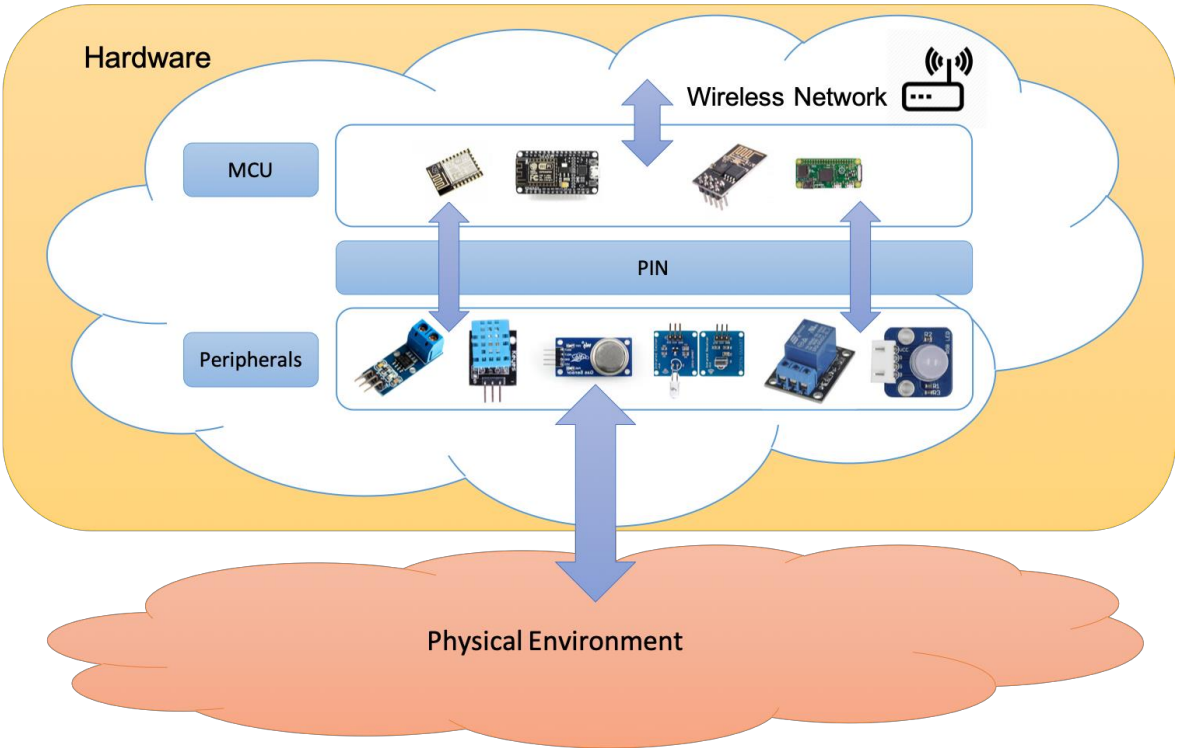

Figura 4 - Hardware utilizzato nel progetto in esame

2.3 Software

La componente software dell'attività progettuale è costituita da un *sistema di regole* e da un *ambiente simbolico*.

Il sistema di regole contiene all'interno della sua base di conoscenza tutti gli aspetti necessari per descrivere la componente hardware e l'ambiente su cui l'hardware viene utilizzato. Al suo interno, la descrizione dell'hardware è effettuata su due piani:

- il primo definisce i vari dispositivi a livello di scheda (*Board Level*); la base di conoscenza, per ogni dispositivo trattato (MCU, sensori, attuatori), contiene la descrizione dei PIN e delle loro funzionalità.

- il secondo definisce le caratteristiche di quel particolare dispositivo in base alle operazioni che può effettuare sull'ambiente.

La descrizione dell'ambiente definisce locazioni, oggetti e stati ambientali. Conoscendo questi aspetti del mondo fisico, il sistema di regole è in grado di legare le varie componenti hardware con quelle dell'ambiente in modo da generare il codice simbolico per la configurazione dei vari dispositivi e per le azioni che essi devono eseguire. Il codice simbolico prodotto viene inviato dal sistema di regole ai nodi tramite un apposito protocollo applicativo.

L'ambiente simbolico è costituito da un *linguaggio ad alto livello* e da un *interprete*. Le componenti lessicali di questo linguaggio sono degli *elementi simbolici* chiamate *parole* (*word*); ogni parola può essere definita facendo corrispondere ad essa una determinata semantica, definita a sua volta tramite altre parole già definite in precedenza. All'interno di questo linguaggio è possibile utilizzare uno o più elementi simbolici, generando stringhe di codice simbolico. L'interprete effettua la traduzione di queste stringhe consentendo la loro esecuzione in tempo reale senza la necessità di effettuare complessi processi di compilazione. L'ambiente simbolico è costruito a partire da istruzioni a basso livello e consente all'utente e al programmatore di interagire con l'hardware sottostante utilizzando un linguaggio ad alto livello. Quindi, grazie all'ambiente simbolico il programmatore può configurare e programmare il nodo usando stringhe di codice simbolico. Allo stesso modo, i nodi della rete possono comunicare tra loro scambiandosi stringhe di codice simbolico.

All'interno dell'attività progettuale il sistema di regole è stato implementato in *Prolog*. Si è scelto questo linguaggio dichiarativo ad alto livello per la sua alta espressività e per la semplicità con cui è possibile ottenere dei risultati a partire dalla descrizione di concetti. Nello specifico si è scelto di utilizzare *SWI Prolog* perché gratuito, disponibile per diverse piattaforme e con diversi moduli implementati. Si è scelto di non implementare l'intero sistema in Prolog perché la sua natura non si presta al controllo diretto dell'hardware, soprattutto in contesti in cui l'hardware è a risorse limitate, per cui l'implementazione non sarebbe stata efficiente.

Come linguaggio per l'ambiente simbolico si è scelto di utilizzare *FORTH*, un linguaggio simbolico imperativo, ad alto livello e basato su *stack*. Tutti i nodi della rete hanno al loro interno un interprete in grado di effettuare la traduzione a *run-time* del codice simbolico in istruzioni da eseguire. Questo tipo di ambiente ha al suo interno un dizionario che contiene le parole definite e può essere esteso con nuove

definizioni in ogni momento; in questo modo è possibile cambiare o incrementare le funzionalità di ogni singolo nodo. La scelta di questo linguaggio, oltre che per la sua facilità d'uso e per l'alta estendibilità delle sue funzionalità, ricade sul fatto che è adatto per *sistemi embedded* e a risorse limitate. Nello specifico, si è scelto di utilizzare *Punyforth*.

Le interazioni tra le componenti software all'interno del progetto in esame sono mostrate in Figura 5.

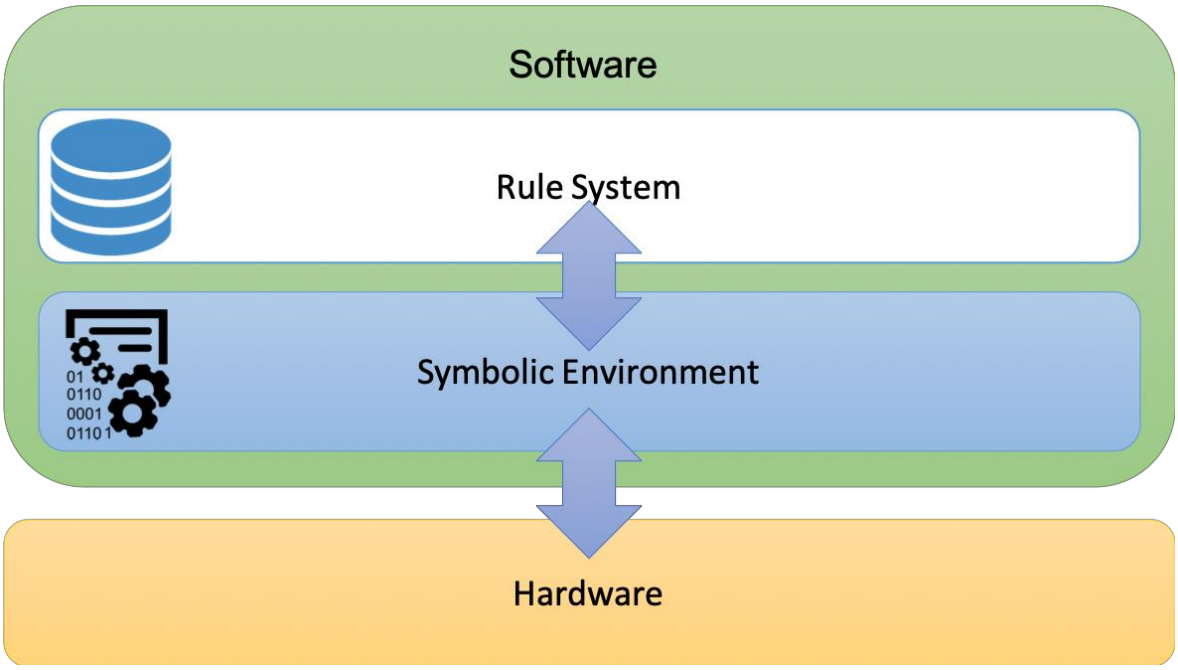

Figura 5 - Interazione tra le componenti software del progetto in esame

Punyforth è un progetto *open source* disponibile su *GitHub* e sviluppato da terzi. Il suo linguaggio è ispirato al linguaggio FORTH; questo ambiente, tramite parole ad alto livello, mette a disposizione del programmatore le funzionalità a basso livello che ogni nodo è in grado di fornire. L'interprete di Punyforth è un'applicazione che gira sul sistema operativo dei vari nodi; nei nodi in cui questo interprete è installato, rende disponibili le funzionalità di rete e quelle di accesso all'hardware tramite un *wrapper*: il suo compito è quello di esporre le funzionalità a basso livello già implementate nell'SDK del sistema operativo del nodo, usando appunto un linguaggio simbolico ad alto livello. Le varie funzionalità sono disponibili sotto forma di moduli che possono essere aggiunti in base alle esigenze. Tra i vari moduli disponibili, questo ambiente dispone di un server TCP REPL attraverso il quale ogni nodo può ricevere ed interpretare stringhe di codice simbolico. In questo modo, da

remoto, si può eseguire la configurazione di ogni nodo e si possono eseguire delle operazioni in base all'obiettivo da raggiungere.

All'interno di Punyforth mancano alcune funzionalità utili per lo scopo che si pone questa attività progettuale. Quindi, oltre alle funzionalità già disponibili, si sono implementati:

- un *client TCP* in modo che ogni nodo possa comunicare con altri nodi indipendentemente da un nodo centralizzato.
- un *client MQTT* per rendere compatibili i nodi con questo protocollo standard diffuso nel contesto dell'IoT.
- un modulo per la memorizzazione permanente in ogni nodo del codice simbolico che definisce la configurazione.
- un protocollo di comunicazione sicuro per lo scambio delle informazioni.

Come si vedrà in dettaglio nel capitolo 6 dedicato all'implementazione, i primi tre punti sono stati implementati direttamente in codice simbolico; il quarto punto è stato implementato integrando i protocolli di criptografia nel sistema operativo dei nodi utilizzando delle librerie scritte in linguaggio C. In Figura 6 è presente uno schema a livelli che mostra le componenti dell'ambiente simbolico.

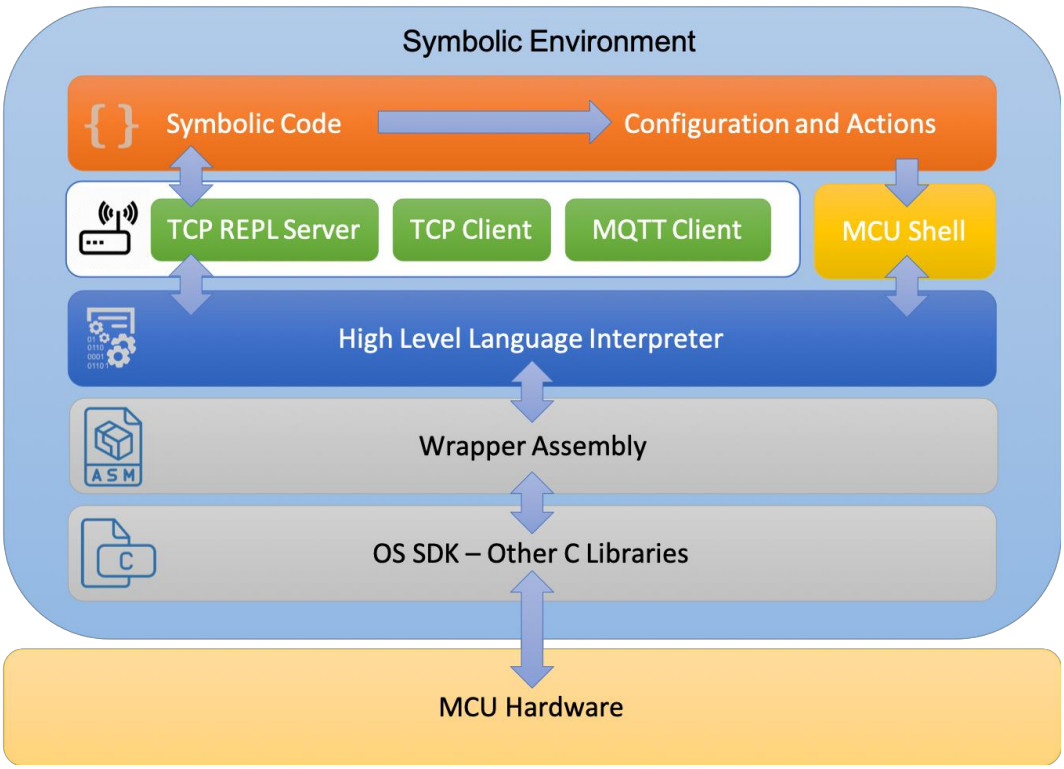

Figura 6 - Schema dell'Ambiente Simbolico utilizzato nel progetto in esame

Nel contesto della domotica, un possibile esempio di attività che può essere svolta dal sistema implementato può essere quello di variare i livelli di luminosità o di temperatura di una determinata stanza. Il sistema cerca di individuare i livelli di luminosità e di temperatura in quel momento presenti nell'ambiente interpellando unicamente i nodi dotati di appositi sensori adatti allo scopo; i dati sensoriali ricevuti vengono confrontati con quelli indicati dall'utente e, nel caso in cui risultassero al di fuori delle soglie impostate, il sistema provvederà alla generazione ed all'invio delle istruzioni ai nodi in grado di agire sugli oggetti capaci di adattare lo specifico parametro fisico alle condizioni desiderate presenti nella sola locazione indicata (lampade e serrande possono agire sul parametro di luminosità, così come condizionatori possono agire sul parametro temperatura e così via).

Il sistema di regole esegue le scelte opportune basando le sue ricerche sulla sua base di conoscenza su cui è presente la descrizione del contesto su cui sta agendo e le possibili azioni che può eseguire. Individuato il percorso per raggiungere l'obiettivo, il sistema di regole contatta i nodi necessari allo scopo utilizzando stringhe di codice simbolico. L'ambiente simbolico dei nodi contattati prende in carico le richieste interpretando a *run-time* le stringhe contenenti le istruzioni da eseguire reiterando il processo finché l'obiettivo non viene raggiunto.

3 ESP8266

Il microcontrollore (MCU) su cui si è portata avanti questa attività progettuale è la *ESP8266-12E*. Prodotto dalla *Espressif Systems*, la ESP8266-12E (Figura 7) fa parte della famiglia delle ESP8266 (Figura 8) e fornisce un sistema su circuito integrato (SoC) Wi-Fi, a basso consumo energetico, dal design compatto e con prestazioni affidabili [14].

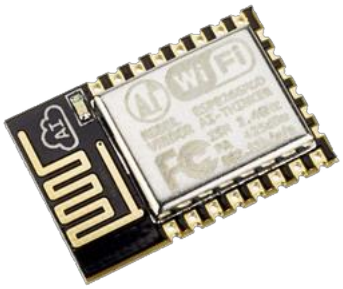

Figura 7 - ESP8266-12E

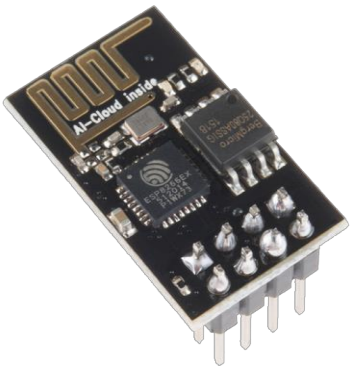

Figura 8 - ESP8266

Sulla scheda della ESP8266 sono presenti diversi *General Purpose Input/Output* (*GPIO*) e delle interfacce *SPI* e *I2C* che le consentono di comunicare con sensori, attuatori e altri dispositivi esterni. Elemento fondamentale per l'attività progettuale svolta, è stata la presenza al suo interno di uno *stack* TCP/IP completo che, assieme al Wi-Fi integrato (vedi Tabella 2 per le specifiche), consente alla ESP8266 di essere usata all'interno di reti TCP/IP, grazie ai quali è possibile l'implementazione e l'uso dei protocolli più comuni nell'ambito dell'IoT e l'utilizzo in campi applicativi come la domotica, reti mesh, prese elettriche ed [15] interruttori intelligenti, videocamere di sorveglianza IP, WSN. Lo sviluppo su questo MCU è facilitato dall'uso di una scheda di sviluppo *NodeMCU Amica* (Figura 9).

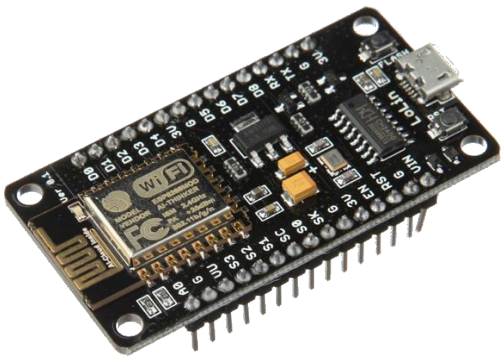

Figura 9 - NodeMCU Amica DevBoard - ESP8266-12E

3.1 CPU

Com'è possibile vedere dalla Tabella 2, la ESP8266 monta un processore *Tensilica L106* (*LX3*) a 32-bit RISC, capace di ottenere bassi consumi e una velocità di 80 Mhz, fino a raggiungere una velocità massima di 160 Mhz.

3.1.1 ISA

L'*Xtensa Instruction Set Architecture* (*ISA*) proposto da *Tensilica* è adatto per i sistemi embedded [15]. Con questo ISA, Tensilica integra nei suoi processori un insieme di strumenti per garantire un'architettura estensibile, configurabile, scalabile, efficiente, a basso costo e a basso consumo; all'interno di questo ISA troviamo istruzioni implementate come istruzioni a 24-bit che consentono di accedere all'intero hardware del processore; le istruzioni che hanno un uso più frequente sono codificate come istruzioni a 16-bit per ridurre la dimensione del codice. L'obiettivo di Tensilica è quello di unire i vantaggi e le prestazioni dei processori RISC con la possibilità di ridurre la dimensione del codice grazie ad uno schema di codifica delle istruzioni con lunghezza variabile, rendendo questi processori principalmente utilizzabili nei sistemi embedded e nei contesti dell'IoT.

Le istruzioni possono accedere a 16 registri *general purpose* attraverso una finestra di scorrimento [16]; la gestione dei registri avviene attraverso la *Register Window* che consente al processore di avere un gran numero di registri fisici accessibili tramite un numero di bit ridotto. Questa tecnica consente di evitare il salvataggio e il ripristino dei registri durante i cambi di procedura o durante gli interrupts.

Categories	Items	Parameters
Wi-Fi	Certification	Wi-Fi Alliance
	Protocols	802.11 b/g/n
	Frequency Range	2.4G ~ 2.5G (2400M ~ 2483.5M)
	Tx Power	802.11 b: +20 dBm
		802.11 g: +17 dBm
		802.11 n: +14 dBm
	Rx Sensitivity	802.11 b: –91 dbm (11 Mbps)
		802.11 g: –75 dbm (54 Mbps)
		802.11 n: –72 dbm (MCS7)
	Antenna	PCB Trace, External, IPEX Connector, Ceramic Chip
Hardware	CPU	Tensilica L106 (LX3) 32-bit processor
	Peripheral Interface	UART/SDIO/SPI/I2C/I2S/IR Remote Control
		GPIO/ADC/PWM/LED Light & Button
	Operating Voltage	2.5V ~ 3.6V
	Operating Current	Average value: 80 mA
	Operating Temperature Range	–40°C ~ 125°C
	Storage Temperature Range	–40°C ~ 125°C
	Package Size	QFN32-pin (5 mm x 5 mm)
	External Interface	-
Software	Wi-Fi Mode	Station/SoftAP/SoftAP+Station
	Security	WPA/WPA2
	Encryption	WEP/TKIP/AES
	Firmware Upgrade	UART Download / OTA (via network)
	Software Development	Supports Cloud Server Development / Firmware and
		SDK for fast on-chip programming
	Network Protocols	IPv4, TCP/UDP/HTTP/FTP
	User Configuration	AT Instruction Set, Cloud Server, Android/iOS App

Tabella 2 - Specifiche ESP8266

3.2 Memoria

Il SoC della ESP8266-12E integra il controllore della memoria e le unità di memoria (SRAM e ROM). La CPU accede alle unità di memoria attraverso un *bus AHB*: la memoria è accessibile effettuando delle richieste e un arbitro deciderà la sequenza di accesso.

Secondo la versione corrente dell'SDK, lo spazio di SRAM disponibile per gli utenti è il seguente:

- Meno di 50 KB quando la ESP8266-12E è in modalità *Station* e connessa ad un router. In questi 50 KB sono considerati accessibili l'heap più la sezione dati.
- La ROM presente nel SoC non è programmabile e il programma utente deve essere memorizzato in una memoria flash SPI esterna.

La ESP8266-12E supporta memorie flash SPI esterne fino a 16 MB di capacità teorica massima. La versione utilizzata nell'attività progettuale svolta ha a bordo 4 MB di memoria flash SPI.

3.3 PIN

Com'è possibile vedere dalla Tabella 3, la ESP8266-12E ha 22 pin sulla scheda. Tra questi, all'interno dell'attività progettuale, sono stati utilizzati:

- *GPIO* (di cui alcuni hanno funzioni multiple).
- *ADC* che converte un segnale analogico in digitale con valori compresi tra 0 e 1024.
- *RXD* e *TXD* per la comunicazione seriale attraverso il *CP2102*, un adattatore da USB a TTL UART.

In Tabella 3 si fornisce una descrizione dei pin presenti sulla scheda della ESP8266-12E [14].

#	Pin Name	Function
1	RST	Reset the module
2	ADC	A/D Conversion result.Input voltage range 0-1v,scope:0-1024
3	EN	Chip enable pin.Active high
4	IO16	GPIO16; can be used to wake up the chipset from deep sleep mode.
5	IO14	GPIO14; HSPI_CLK
6	IO12	GPIO12; HSPI_MISO
7	IO13	GPIO13; HSPI_MOSI; UART0_CTS
8	VCC	3.3V power supply (VDD)
9	CS0	Chip selection
10	MISO	Salve output Main input
11	IO9	GPIO9
12	IO10	GPIO10
13	MOSI	Main output slave input
14	SCLK	Clock
15	GND	GND
16	IO15	GPIO15; MTDO; HSPICS; UART0_RTS
17	IO2	GPIO2; UART1_TXD
18	IO0	GPIO0
19	IO4	GPIO4
20	IO5	GPIO5
21	RXD	UART0_RXD; GPIO3
22	TXD	UART0_TXD; GPIO1

Tabella 3 - Pin ESP8266-12E

3.3.1 GPIO

Le 17 GPIO a disposizione presenti nella scheda della ESP8266-12E possono avere assegnate diverse funzioni. Ogni GPIO può essere configurata con un *pull-up* o un *pull-down* interno o impostata ad alta impedenza. Quando una GPIO viene configurata come input, il dato viene memorizzato in alcuni registri software. Inoltre l'input può essere anche configurato per inviare interrupts secondo le modalità *edge trigger* o *level trigger*.

Per le operazioni a basso consumo, le GPIO possono mantenere impostato il loro stato corrente; al loro spegnimento, tutti i segnali di output impostati sono portati al segnale basso.

Le GPIO possono essere configurate per usare le altre funzioni come *I2C*, *I2S*, *UART*, *PWM*, ecc.

3.3.2 Serial Peripheral Interface (SPI/HSPI)

La ESP8266-12E ha tre SPI:

- *General Slave/Master SPI*
- *Slave SDIO/SPI*
- *General Slave/Master HSPI*

La modalità SPI può essere implementata attraverso la programmazione software raggiungendo al massimo una frequenza di clock di 80 MHz.

3.3.3 I2C Interface

La ESP8266-12E ha una I2C che può essere usata per la connessione con altri MCU e altre periferiche (sensori, attuatori, display). La ESP8266-12E supporta entrambe le modalità Master/Slave I2C; queste modalità possono essere realizzate attraverso la programmazione software raggiungendo al massimo una frequenza di clock di 100 KHz.

3.3.4 I2S Interface

La ESP8266-12E ha un'interfaccia I2S per i dati di input e un'altra per i dati di output. Questo tipo di interfaccia è generalmente usata per la raccolta e l'elaborazione di dati, la trasmissione di dati audio e, più in generale, per l'input e l'output di dati su un canale seriale. La modalità I2S può essere implementata attraverso la programmazione software.

3.3.5 Universal Asynchronous Receiver Transmitter (UART)

La ESP8266-12E ha due interfacce *UART*:

- *UART0*: supporta sia la ricezione (Rx) che la trasmissione (Tx) di segnali ed è usata per le comunicazioni seriali.
- *UART1*: supporta solo le trasmissioni di segnali.

La trasmissione dei dati attraverso queste interfacce raggiunge i 115200 baud (4.5 Mbps). Il tasso di trasferimento dipende dalla frequenza dell'oscillatore: se la frequenza è impostata a 40 MHz, il trasferimento avviene a 115200 baud; se è

impostata a 26 MHz, il trasferimento avviene a 74800 baud. All'avvio della ESP8266-12E, sulla UART0 vengono trasmesse alcune informazioni.

La UART0, oltre che per la comunicazione seriale, può essere usata anche per il download delle immagine binarie da memorizzare all'interno della memoria flash esterna.

3.3.6 Pulse-Width Modulation (PWM)

La ESP8266-12E ha 4 interfacce *PWM*. Questa interfaccia è molto utile per controllare dispositivi come luci LED RGB, *buzzer*, motori passo passo, ecc. . Questa funzionalità può essere implementata attraverso la programmazione software.

3.3.7 IR Remote Control

La ESP8266-12E ha un'interfaccia per implementare il controllo remoto tramite infrarossi. Possiede un PIN su cui è possibile implementare la funzionalità di ricezione e un PIN in cui è possibile implementare la funzionalità di trasmissione. Questa interfaccia usa la codifica *NEC* per la modulazione/demodulazione a 38 KHz.

3.3.8 Analog-to-Digital Converter (ADC)

La ESP8266-12E possiede un PIN per la conversione di un segnale analogico in un segnale digitale con una precisione di 10 bit (con valori quindi da 0 a 1023). Questa interfaccia e molto utile per leggere i dati dai sensori analogici.

3.4 Wi-Fi

La ESP8266-12E ha un ricetrasmettitore RF capace di ricevere e trasmettere alla frequenza di 2.4 GHz e supporta i canali che vanno dall'8 al 14 secondo lo standard IEEE802.11 b/g/n. Al suo interno sono implementati i protocolli TCP/IP, WLAN MAC 802.11 b/g/n e Wi-Fi Direct. Secondo quanto specificato all'interno dell'SDK, la ESP8266-12E supporta le modalità Wi-Fi *Acces Point* (WIFI_AP), *Station* (WIFI_STA) e *disattivato* (WIFI_OFF).

4 Ambiente Simbolico

Come detto nella sezione 2.3, si è scelto di utilizzare Punyforth per l'ambiente simbolico. Punyforth è un linguaggio ispirato al linguaggio FORTH che ha come obiettivo primario l'utilizzo nel contesto dell'*Internet of Things* e nei dispositivi a basso costo, a basso consumo e a risorse limitate come la ESP8266. L'interprete di Punyforth, oltre che sulla ESP8266, gira anche su architetture x86 (Linux) e ARM (Raspberry PI).

Secondo quanto affermato dal suo sviluppatore, Punyforth ha come obiettivo la semplicità di programmazione, l'alta interattività, la possibilità di essere esteso, il basso consumo di memoria e l'efficienza di uso delle risorse a disposizione. Parte di queste caratteristiche (la semplicità di programmazione, l'alta interattività e l'alta estendibilità), Punyforth le eredita dal linguaggio FORTH.

Punyforth è un linguaggio imperativo, basato su *stack*, che fornisce un ambiente simbolico interattivo accessibile tramite una *shell REPL* (*Read Eval Print Loop*) e che mette a disposizione del programmatore i vantaggi della programmazione dei linguaggi compilati con quelli della programmazione dei linguaggi interpretati.

All'interno di questo capitolo si descrivono le caratteristiche e le funzionalità di Punyforth, già implementate direttamente dal suo sviluppatore, che sono state fondamentali per lo svolgimento di questa attività progettuale.

4.1 Stack

Essendo basato su *stack*, Punyforth non ha variabili locali: gli operandi sono impilati nello *stack*, da cui sono consumati in base all'ordine di inserimento e in base alla quantità richiesta dell'operatore attualmente richiamato, secondo la cosiddetta *Reverse Polish Notation* (*RPN*) o *Notazione Postfissa*. All'interno di Punyforth, come su FORTH, sono presenti due *stack*:

- *Data Stack*: memorizza i dati da utilizzare tra più parole e ogni parola può eseguire le classiche operazioni di *push* e *pop* all'interno di esso.
- *Return Stack*: memorizza i dati e le informazioni tra le varie *subroutine* annidate che vengono richiamate.

Un semplice esempio di uso del *Data Stack* è il seguente: inserendo all'interno della *shell* la stringa contenente il codice simbolico `1 2 +`, l'interprete di Punyforth

inserisce i numeri `1` e `2` all'interno del *Data Stack*, esegue la parola `+` che, a sua volta, consuma i due elementi in cima al *Data Stack*, calcolando la loro somma ed inserendo il risultato di questa operazione all'interno del *Data Stack*.

Per fare in modo che il *Data Stack* contenga esattamente ciò che serve e nel giusto ordine per gli usi futuri, questo può essere manipolato attraverso alcune parole che consentono di modificare la posizione e il numero degli elementi presenti al suo interno. Queste parole si suddividono in tre categorie:

- *Parole di rimozione*: rimuovono 1 o n elementi dal *Data Stack*.
- *Parole di duplicazione*: duplicano 1 o n elementi all'interno del *Data Stack*.
- *Parole di permutazione*: scambiano 1 o n elementi all'interno del *Data Stack*.

4.2 Dizionario

Tutte le parole sono memorizzate all'interno di un *dizionario*. Questo dizionario mappa la parole con il loro codice simbolico eseguibile compilato o le strutture dati create.

Usando la parola `:` (*colon*), il programmatore può definire nuove parole usate per identificare il codice simbolico eseguibile, estendendo così il dizionario con nuove definizioni chiamate *Colon Definition*. La parola `:` indica all'interprete che sta iniziando una nuova definizione e che quindi deve uscire dalla modalità chiamata *Run Time* per entrare nella modalità chiamata *Compile Time*. Per indicare all'interprete che la definizione è terminata e che può compilare il codice simbolico, creando quindi un nuovo elemento all'interno del dizionario, si usa la parola `;` (*semicolon*) che termina la modalità di *Compile Time* e riporta l'interprete in modalità *Run Time*. La forma utilizzata per definire nuove parole è la seguente:

```
: <header> <body> ;
```

Ad esempio, nel contesto dell'IoT, si potrebbe definire la parola ON in questo modo:

```
: ON GPIO_HIGH gpio-write ;
```

dove `GPIO_HIGH` è la costante che contiene il valore alto per l'MCU che si sta utilizzando e `gpio-write` è la parola che fa riferimento al codice eseguibile da eseguire per impostare una GPIO con quel valore. Per cui, supponendo che la costante `LAMP` contenga l'identificativo del PIN a cui è collegato il *relay* per accendere una lampada, per esempio `15`, eseguendo

```
LAMP ON
```

l'interprete inserisce l'intero `15` all'interno del *Data Stack* ed esegue la parola `ON` sopra definita che, a sua volta, impila il valore di `GPIO_HIGH` all'interno del *Data Stack* e richiama il codice eseguibile della `gpio-write` che consuma i due valori presenti in cima al *Data Stack*, impostando così il valore alto nel PIN 15, inviando il segnale al *relay* collegato e accendendo così la lampada.

Le *Colon Definition*, una volta compilate, non usano direttamente le parole del codice simbolico utilizzate al loro interno durante la loro definizione, ma usano il loro *Execution Token* (xt), cioè l'indirizzo di memoria a cui fa riferimento la *Colon Definition* dopo essere stata compilata. In questo modo, qualora il programmatore volesse ridefinire una parola già definita all'interno del dizionario, può farlo tranquillamente senza stravolgere il comportamento delle parole che in quel momento utilizzano la parola che si sta andando a ridefinire. Attraverso la parola `'` (*tick*) è possibile ottenere *l'Execution Token* di una parola.

Qualora invece il programmatore volesse modificare in modo assoluto una parola già definita, riportando quindi il nuovo comportamento anche nelle altre parole già definite in cui la parola stessa è già stata utilizzata, può usare le *Deferred Word*. Una *Deferred Word*, in Punyforth, si dichiara attraverso la parola

```
defer: <nome della deferred word>
```

e il suo comportamento può essere modificato attraverso

```
' <nome della deferred word> is: <nome della parola con il
Nuovo comportamento>
```

Bisogna fare attenzione nel caso in cui si volesse ridefinire una parola riutilizzandola all'interno del suo nuovo codice simbolico. Di seguito un esempio:

```
: ON GPIO_HIGH gpio-write ;

: ON "Accensione in corso" type ON ;
```

un caso di questo tipo, genererebbe una ricorsione infinta poiché la parola `ON` all'interno della nuova definizione farebbe riferimento alla nuova definizione stessa. Per consentire questa ridefinizione senza incorrere nella ricorsione, si usa la parola `override`

```
: ON override "Accensione in corso" type ON ;
```

In questo modo la parola `ON`, all'interno del codice simbolico della ridefinizione, farà riferimento alla definizione precedente.

Altre parole utilizzate per le definizioni sono quelle per le strutture dati. È possibile definire variabili e costanti utilizzando le parole `variable:` e `init-variable:` per le variabili e `constant:` per le costanti.

Come si era detto in precedenza, vi sono due modalità di esecuzione: se si volesse eseguire una parola quando l'interprete è in *Compile Time*, si possono usare le *Immediate Word*. Per indicare che una parola è *Immediate*, si usa la parola `immediate` dopo la parola `;`, quindi alla fine della sua definizione.

4.3 Strutture di Controllo

Come FORTH, Punyforth supporta le parole per le strutture di controllo condizionali e iterative. Le strutture di controllo sono parole che non hanno una semantica d'interpretazione e non possono essere usate quando l'interprete è in modalità *Run Time* ma solo quando è in modalità *Compile Time*, come, ad esempio, all'interno delle definizioni di nuove parole.

4.3.1 Strutture di Controllo Condizionali

Come FORTH, Punyforth supporta le parole per le strutture condizionali. Di seguito l'elenco di queste.

4.3.1.1 If Else Then

Punyforth supporta il costrutto `if else then` del FORTH attraverso il quale è possibile effettuare dei salti condizionali in base al risultato della condizione booleana:

```
<bool> if <consequent> else <alternative> then
```

La parte `else` non è obbligatoria e, nei casi in cui non serve, può essere omessa.

4.3.1.2 Switch Case

Punyforth supporta il costrutto `switch-case` secondo questa modalità:

```
case
    <caseA> of <action> endof
    <caseB> of <action> endof
    <caseN> of <action> endof
```

```
        <default action>
    endcase
```

La parola `case` consuma il valore presente in cima al *Data Stack* e lo confronterà con i vari casi presenti all'interno del costrutto, eseguendo l'azione corrispondente in caso di corrispondenza o quella di `default` se il valore non rientra in nessuno dei casi specificati.

4.3.2 Strutture Condizionali Iterative

Punyforth supporta le strutture condizionali iterative attraverso le quali è possibile ripetere N o infinte volte la stessa porzione di codice simbolico.

4.3.2.1 Do Loop

Questo tipo di ciclo effettua l'iterazione del codice simbolico presente all'interno di `<loop-body>` partendo da un intero iniziale `<start>` e ripetendo il codice simbolico finché l'indice del ciclo non raggiungere un certo limite `<limit>`:

```
<limit> <start> do <loop-body> loop
```

L'indice del ciclo è accessibile attraverso la parola `i` che può essere richiamata all'interno del `<loop-body>` e, se richiamata, inserisce il contatore in cima al *Data Stack*. L'indice è memorizzato all'interno del *Return Stack*. Qualora ci fosse un ciclo annidato, il ciclo più interno può accedere all'indice del ciclo più esterno attraverso la parola `j`.

All'interno di Punyforth è presente una versione diversa della parola `loop`, la `+loop`, che consente al programmatore di specificare l'incremento (o il decremento, se negativo) `<increment>` dell'indice.

```
<limit> <start> do <loop-body> <increment> +loop
```

È possibile uscire dal ciclo `do loop` usando le parole `unloop` per pulire il *Return Stack* e `exit`.

4.3.2.2 Begin Until

Questo tipo di ciclo effettua l'iterazione del codice simbolico presente all'interno di `<loop-body>` ripetendo, almeno una volta, il codice simbolico finché la condizione `<bool>` è vera

```
begin <loop-body> <bool> until
```

Se scritto nella forma

```
begin <loop-body> again
```

il ciclo sarà infinito.

4.3.2.3 While

Questo tipo di ciclo è simile al `begin until`, soltanto che la prima esecuzione del codice simbolico all'interno di `<loop-body>` avviene solo se la condizione `<bool>` è vera

```
begin <bool> while <loop-body> repeat
```

È possibile uscire dal ciclo direttamente usando la parola `exit`; non c'è bisogno di utilizzare la parola `unloop` perché il `while` non usa il *Return Stack*.

4.4 Eccezioni

Nel caso in cui una parola entrasse in una condizione di errore, Punyforth consente di lanciare un'eccezione che può essere catturata e gestita all'esterno della parola stessa. Un'eccezione viene dichiarata attraverso la parola

```
exception: <exception name>,
```

lanciata attraverso la parola

```
<exception name> throw
```

e catturata attraverso la parola

```
['] <parola che può lanciare un'exception> catch.
```

La parola `catch` lascia in cima al *Data Stack* il riferimento all'eccezione catturata che può essere analizzata dai costrutti condizionali eseguendo l'azione appropriata.

4.5 Quotation e Combinator

Come detto precedentemente, in Punyforth sono presenti diverse parole per la manipolazione del *Data Stack*. Un eccessivo utilizzo di queste parole può rendere difficile la manutenzione e la lettura del codice simbolico presente all'interno delle parole definite dal programmatore. Un modo per rendere migliore la scrittura del codice, può essere quello di usare le *Quotation* e i *Combinator*.

4.5.1 Quotation

Una *Quotation* è una parola anonima, simile alle *lambda expression* presenti in altri linguaggi di programmazione. La parola `{` è usata per indicare all'interprete che sta iniziando la modalità *Compile Time* del codice simbolico della *Quotation* all'interno della definizione della parola corrente; la parola `}` è usata per indicare all'interprete che è terminata la modalità *Compile Time* della *Quotation*. Di seguito l'esempio della *Colon Definition* `BLINK`, il cui compito è quello di far lampeggiare una lampada o un led:

```
: BLINK "Start flashing" type { dup ON 500 ms OFF 500 ms } ;
```

Quando l'interprete riceve, ad esempio, in input la stringa di codice simbolico

```
LAMP BLINK 3 TIMES
```

inserisce in cima al *Data Stack*, in questo ordine: il valore del PIN a cui è collegato il *relay* della lampada, l'indirizzo di memoria in cui è stata collocata la stringa `"Start flashing"` (che è immediatamente consumato dalla parola `type` che si occupa di stampare la stringa nell'*output stream*), l'*Execution Token* della *Quotation* e, infine, il numero `3`. La parola `TIMES` è un *Combinator* che si occupa di consumare i tre elementi presenti in cima al *Data Stack*, ripetendo l'operazione della *Quotation* 3 volte sul PIN `LAMP`, facendo quindi lampeggiare la lampada per 3 volte.

4.5.2 Combinator

Un *Combinator* è una parola definita nel dizionario che consuma una *Quotation* e 1 o N elementi dal *Data Stack*. La parola `TIMES` dell'esempio precedente può essere definita in questo modo:

```
: TIMES 0 do dup >r keep r> loop 2drop ;
```

Facendo riferimento al codice simbolico del paragrafo precedente, passo dopo passo, la parola `TIMES`:

- consuma il numero `3` presente in cima al *Data Stack* e lo usa come `<limit>` del ciclo `do loop`
- duplica l'*Execution Token* della *Quotation* di `BLINK` (che si trova in cima al *Data Stack*)
- il primo *Execution Token*, viene inserito all'interno del *Return Stack* da `>r`
- il secondo *Execution Token*, viene utilizzato da `keep`, un altro *Combinator*; questo *Combinator* (che consuma dal *Data Stack* una *Quotation* e un elemento) ha il compito di eseguire la *Quotation* utilizzando il suo *Execution*

Token immediatamente disponibile in cima al *Data Stack* e l'elemento immediatamente precedente, per poi ripristinarlo alla fine dell'esecuzione della *Quotation*; `keep`, quindi, esegue la *Quotation* della parola `BLINK` sul PIN `LAMP` e, al suo termine, ripristina in cima al *Data Stack* il PIN `LAMP`

- `r>` ripristina l'*Execution Token* della *Quotation* per essere usata nel prossimo ciclo
- il ciclo si ripete fin quando non si raggiunge il `<limit>`
- alla fine del ciclo, viene usato il `2drop` per cancellare dal *Data Stack* i due elementi rimasti dopo l'esecuzione del ciclo `do loop` (il valore di `LAMP` e l'*Execution Token* della *Quotation*) riportando così il *Data Stack* nelle condizioni in cui era prima dell'esecuzione del codice simbolico di esempio.

4.6 Create: e Does>

Le parole `create:` e `does>` consentono al programmatore di combinare una struttura dati con un'azione.

La parola `create:` è come la parola `:`, cioè consente di creare nuove voci nel dizionario ma, a differenza di questa, crea una voce al suo interno con un `<header>` senza `<body>`. Questa parola, legge dall'input stream il nome della nuova definizione che si sta andando a creare, generando una nuova voce nel dizionario per la struttura dati.

La parola `does>`, specificata all'interno della definizione, rende possibile (a *Run Time*) l'esecuzione dell'azione specificata all'interno della definizione e quella immediatamente dopo di essa. Di seguito un esempio che mostra com'è stata definita la parola `constant:`

```
: constant: create: , does> @ ;

15 constant: LAMP
```

Quando si esegue il codice simbolico per creare la costante `LAMP` con valore `15`:

- il valore `15` viene inserito in cima al *Data Stack*
- viene invocata la parola `constant:`
- al suo interno, viene utilizzata la parola `create:` che legge i caratteri della parola `LAMP`, creando la voce all'interno del dizionario

- la parola `,` si occupa di memorizzare il valore `15` all'interno dell'area di memoria puntata della costante `LAMP`
- la parola `does>` istruirà l'interprete su cosa fare quando la parola `LAMP` sarà usata a *Run Time*. Nell'esempio in questione, l'azione che dovrà essere eseguita è quella della parola `@` il cui compito è quello di inserire in cima al *Data Stack* il contenuto della costante.

Un altro esempio d'uso più complesso potrebbe essere quello della creazione di una parola per creare e gestire gli *array*:

```
: array: create: cells allot does> swap cells + ;

4 array: PIN
```

La parola `array:`, come la parola `constant:` dell'esempio precedente, si occupa di creare una nuova voce nel dizionario, con in più il compito di riservare anche dello spazio all'interno della memoria tramite le parole `cells allot`. L'azione che sarà eseguita a *Run Time* è specificata dopo `does>`, cioè `swap cells +`: il suo compito è semplicemente quello di lasciare all'interno del *Data Stack* l'indirizzo di memoria della posizione desiderata dell'*array*. Ad esempio, se il programmatore volesse memorizzare il PIN `LAMP` all'interno della posizione 0 dell'*array* `PIN`, deve eseguire `LAMP 0 PIN !`; se volesse memorizzare il PIN LED all'interno della posizione 1 dell'*array*, deve eseguire `LED 1 PIN !`.

4.7 ESP8266

Per quanto riguarda la ESP8266, Punyforth ha delle implementazioni aggiuntive rispetto alle altre architetture per cui è stato sviluppato. La sua struttura implementativa, in riferimento alla ESP8266, sarà analizzata nel capitolo 5. In questa sezione saranno mostrate le parole che consentono di sfruttare le funzionalità che offre la ESP8266. Alcune di queste parole sono state definite all'interno dell'immagine binaria di Punyforth attraverso dei file contenenti codice sorgente *Assembly* per l'architettura del Tensilica L106 (in questi casi il codice *Assembly* è stato usato come un *wrapper*, in modo da esporre all'interno del dizionario alcune funzioni presenti all'interno dell'SDK del sistema operativo), altre parole invece sono state definite all'interno dei moduli (file contenenti stringhe rappresentanti codice simbolico) che, se utili al programmatore o all'utente, devono essere caricati all'interno della flash SPI (come si vedrà nella sezione 5.3.1), per poi essere caricati dall'applicazione Punyforth attraverso l'apposita parola `LOAD`.

4.7.1 Wi-Fi

Come descritto nella sezione 3.4, la ESP8266 supporta le modalità Wi-Fi *AP* (*Access Point*) e *Station* (*Client Wireless*). Alcune parole che consentono di utilizzare le varie funzionalità legate al Wi-Fi si trovano già all'interno dell'immagine binaria, altre all'interno del modulo *wifi.forth* (consultabile alla sezione 9.1.1).

Per quanto riguarda le modalità Wi-Fi e le modalità di autenticazione e cifratura supportate, all'interno di questo modulo troviamo definite alcune costanti che memorizzano i vari codici assegnati dall'SDK del sistema operativo della ESP8266, in modo da avere dei *token* mnemonici di codice simbolico da utilizzare all'interno delle definizioni o delle stringhe di codice simbolico eseguibile. Troviamo poi le parole:

- `wifi-connect password ssid -- | throws:EWIFI )`: prima di tutto setta la modalità Wi-Fi *Station* tramite la costante `STATION_MODE`; successivamente esegue la parola `wifi-set-station-config` che consuma dal *Data Stack* la `password` e l'`ssid` della rete Wi-Fi a cui ci si vuole connettere; queste due stringhe vengono memorizzate all'interno della flash SPI (esattamente all'interno dell'ultimo blocco di flash disponibile) e sono utilizzati dalla parola `wifi-station-connect` per effettuare infine la connessione.

  Le parole `wifi-set-mode`, `wifi-set-station-config`, `wifi-station-connect` sono definite in *Assembly* e la loro funzione corrispondente dell'SDK prevede il ritorno di un codice di stato che viene inserito in cima al *Data Stack*; la parola `check-status` si occupa di verificare che questo codice non sia diverso da 1, caso in cui lancerà l'eccezione `EWIFI`.

- `wifi-softap ( max-connections channels hidden authmode password ssid -- | throws:EWIFI )`: per prima cosa setta la modalità Wi-Fi *AP* tramite la costante `SOFTAP_MODE`; successivamente esegue la parola `wifi-set-softap-config` che consuma dal *Data Stack* i seguenti valori `max-connections channels hidden authmode password ssid` attraverso i quali è possibile impostare, rispettivamente, il numero massimo di *Client Station* ammessi, il canale, se l'SSID sarà visibile o meno, la modalità di autenticazione e la cifratura da utilizzare, la password e l'SSID dell'*AP*. L'`authmode` può avere i valori memorizzati all'interno delle costanti `AUTH_OPEN`, `AUTH_WEP`, `AUTH_WPA_PSK`, `AUTH_WPA2_PSK`,

`AUTH_WPA_WPA2_PSK` e `AUTH_MAX`. `password` e `ssid` sono gli indirizzi di memoria in cui sono posizionate le rispettive stringhe.

La parola `wifi-set-softap-config` è stata definita in *Assembly* e consuma tutti i parametri dal *Data Stack* per poi passarli alla funzione corrispondente definita all'interno dell'SDK.

Questa parola imposta la modalità Wi-Fi della ESP8266 in *AP*. Può essere necessario impostare manualmente un indirizzo IP da assegnare alla ESP8266; questa operazione è possibile attraverso la parola `wifi-set-ip` che consuma l'indirizzo IP (sotto forma di intero) dal *Data Stack*. Se nella rete non è presente un server *DHCP* per l'assegnazione automatica degli indirizzi IP e dei vari parametri di rete ai *Client Station* che si collegano all'*AP* creato dalla ESP8266, è possibile usare la ESP8266 stessa come server *DHCP*: la parola `dhcpd-start` consente di avviare un server *DHCP* ricevendo come parametri `max_leases` e `first_ip` i quali impostano, rispettivamente, il numero massimo di indirizzi IP che il server può rilasciare e l'indirizzo IP di partenza. Di seguito un esempio che mostra l'uso completo della modalità *AP*:

```
172 26 93 1 >ipv4 wifi-set-ip

4 3 0 AUTH_WPA2_PSK "mypasswd" "myssid" wifi-softap

8 172 26 93 2 >ipv4 dhcpd-start
```

la parola `>ipv4` si occupa di convertire l'indirizzo IP da 4 ottetti in un numero intero. Le parole `wifi-set-ip` e `dhcpd-start` sono state definite in *Assembly* e richiamano le corrispettive funzioni presenti all'interno dell'SDK.

4.7.2 GPIO, ADC e PWM

Un altro aspetto fondamentale nell'ambito dell'IoT è la possibilità di controllare i PIN GPIO e i PIN con funzionalità di ADC e PWM, per potere comunicare con tutti i possibili dispositivi (sensori e attuatori). All'interno di Punyforth, sono state definite delle parole per fare ciò:

- `gpio-mode (pin mode -- )`: abilita la GPIO del PIN `pin` alla modalità `mode`; `mode` può assumere i valori memorizzati all'interno delle costanti `GPIO_IN`, `GPIO_OUT` e `GPIO_OUT_OPEN_DRAIN`.
- `gpio-write (pin value -- )`: imposta il valore `value` al PIN `pin`; `value` può assumere i valori memorizzati all'interno delle costanti `GPIO_HIGH` e `GPIO_LOW`.

- `gpio-read (pin -- value)`: legge il valore `value` dal PIN `pin` e lo posiziona in cima al *Data Stack*.
- `adc-read ( -- value)`: legge il valore `value` dal PIN ADC e lo posiziona in cima al *Data Stack*.
- `pwm-init (pins length -- )`: abilita alla modalità PWM i PIN `pins`; `pins` è un'*array* di lunghezza `length`.
- `pwm-freq (freq -- )`: imposta la frequenza `freq` usata dal PWM. Il valore ammesso è un intero da 16 bit.
- `pwm-duty (duty -- )`: imposta la durata `duty` del *Duty Cycle* usato dal PWM. Il valore ammesso è un intero da 16 bit.
- `pwm-start ( -- )`: avvia l'esecuzione del PWM.
- `pwm-stop ( -- )`: interrompe l'esecuzione del PWM.

Tutte le parole illustrate sopra sono state definite in *Assembly* poiché usano le funzioni corrispondenti presenti all'interno dell'SDK. Le varie costanti sono state definite all'interno del modulo *gpio.forth*.

Di seguito alcuni esempi d'uso:

```
\ GPIO:
PIN GPIO_OUT gpio-mode
PIN GPIO_HIGH gpio-write

\ ADC
adc-read

\ PWM per RGB LED
12 constant: RED
13 constant: GREEN
14 constant: BLUE
create: PINS RED c, GREEN c, BLUE c,
PIN GPIO_OUT gpio-mode
```

```
PINS 3 pwm-init
2000 pwm-freq
1000 pwm-duty
pwm-start
```

4.7.3 Network

In questa sezione si vedranno le parole disponibili per potere eseguire dei client e dei server TCP e/o UDP usando direttamente stringhe di codice simbolico. Questo aspetto è molto importante perché attraverso queste parole si ha la possibilità di scrivere protocolli applicativi in codice simbolico per potere utilizzare servizi già esistenti (o servizi personalizzati) all'interno della rete. Le parole interessate in questo contesto sono state definite all'interno del modulo *netcon.forth* (consultabile alla sezione 9.1.2).

- `netcon-new ( type -- netcon | throws:ENETCON )`: il suo compito è quello di aprire un nuovo *socket*. Questa parola esegue l'*override* della parola omonima definita in *Assembly* che richiama la funzione corrispondente dell'SDK e consuma dal *Data Stack* il tipo `type` (che può essere `TCP` o `UDP` definiti tramite le apposite costanti) per aprire un nuovo *socket*. La funzione crea la `struct netcon` del linguaggio C in cui sono presenti diversi parametri, tra cui il *socket* stesso. L'indirizzo di memoria in cui è stato memorizzato il *socket* viene rilasciato in cima al *Data Stack*. L'*override* è stato eseguito per potere effettuare altre operazioni oltre a quelle previste dall'SDK: se la `netcon-new` definita in *Assembly* non riesce ad aprire un nuovo *socket*, rilascia in cima al *Data Stack* il valore `0`; nel caso in cui si presenta questa condizione, la `netcon-new` ridefinita all'interno del modulo lancia l'eccezione `ENETCON`; se non si presenta la condizione precedente, la parola ridefinita imposta il *timeout* per la ricezione dei dati nel *socket* aperto usando la parola `netcon-set-recvtimeout` che consuma dal *Data Stack* l'indirizzo di memoria del *socket* e la costante `RECV_TIMEOUT_MSEC`; al termine della parola ridefinita, l'indirizzo di memoria del *socket* aperto `netcon` viene lasciato in cima al *Data Stack* a disposizione del programmatore per le operazioni future.

- `netcon-connect ( port host type -- netcon | throws:ENETCON )`: il suo compito è quello di connettersi ad un *socket* TCP o UDP. Esegue l'*override* della parola omonima definita in *Assembly* che utilizza la funzione corrispondente dell'SDK e consuma dal *Data Stack*: l'indirizzo di memoria del *socket* aperto dalla `netcon-new` (di tipo `type` in base al valore presente in cima al *Data Stack*), l'indirizzo di memoria della stringa `host` contenente l'*host* a cui ci si vuole connettere (può essere sia un indirizzo IP che un nome di dominio) e l'intero `port` che rappresenta la porta. Se la risoluzione del nome di dominio o la connessione falliscono, la funzione definita in linguaggio C restituisce un intero rappresentante l'errore che viene analizzato dalla parola `check` presente all'interno della parola ridefinita lanciando, se il caso, l'eccezione `ENETCON`. Al termine della parola ridefinita, l'indirizzo di memoria del *socket* aperto `netcon` viene lasciato in cima al *Data Stack*.
- `netcon-bind ( port host netcon -- | throws:ENETCON )`: il suo compito è quello di effettuare l'associazione tra il *socket* aperto, l'indirizzo e la porta della ESP8266. È usata nella creazione dei server TCP o UDP. Esegue l'*override* della parola omonima definita in *Assembly*, che richiama la funzione corrispondente dell'SDK; questa parola consuma: l'indirizzo di memoria del *socket* `netcon`, l'indirizzo di memoria della stringa `host` e l'intero della porta `port`; se la risoluzione del nome di dominio o il *binding* falliscono, la funzione definita in linguaggio C restituisce un intero rappresentante l'errore che viene analizzato dalla parola `check` presente all'interno della parola ridefinita lanciando, se il caso, l'eccezione `ENETCON`.
- `netcon-listen ( netcon -- | throws:ENETCON )`: è usata con i *socket* server TCP per mettere il server in attesa di ascolto dei client. Esegue l'*override* della parola omonima definita in *Assembly*, che richiama la funzione corrispondente dell'SDK; questa parola consuma soltanto l'indirizzo di memoria del *socket* `netcon` su cui rimanere in attesa; la funzione definita all'interno dell'SDK restituisce un intero che poi viene analizzato dalla parola `check` presente all'interno della parola ridefinita lanciando, se il caso, l'eccezione `ENETCON`.
- `netcon-tcp-server ( port host -- netcon | throws:ENETCON )`: crea un server TCP utilizzando le parole illustrate in precedenza effettuando il *binding* sui parametri `host` e `port` presenti in cima al *Data Stack*, mettendo così in ascolto il *socket*. In caso di successo, rilascia in cima

al *Data Stack* l'indirizzo di memoria del *socket* server `netcon`, altrimenti viene lanciata l'eccezione `ENETCON`.

- `netcon-udp-server ( port host -- netcon | throws:ENETCON )`: come la `netcon-tcp-server` solo che crea un server UDP e non mette il *socket* in ascolto.
- `netcon-accept ( netcon -- new-netcon | throws:ENETCON )`: è usata nel caso di *socket* server TCP per accettare le connessioni provenienti dai client. Questa parola esegue l'*override* della parola omonima definita in *Assembly*, che richiama la funzione corrispondente dell'SDK; questa parola consuma dal *Data Stack* l'indirizzo di memoria del *socket* `netcon` e restituisce la *struct* `accept_res` del linguaggio C contenente il codice di stato e l'indirizzo di memoria della *struct* `netcon` in cui è presente il *socket* aperto per il client; la parola omonima definita in *Assembly* effettua l'inserimento in cima al *Data Stack* di questi due valori. All'interno della parola ridefinita è presente un ciclo `begin again` che esegue la parola `netcon-accept` definita in *Assembly* ripetutamente ogni volta che si presenta l'errore di *timeout* `NC_ERR_TIMEOUT`; se non c'è stato un *timeout* e non ci sono stati problemi nell'apertura del *socket* (che hanno lanciato la solita eccezione `ENETCON`), il ciclo viene interrotto tramite la parola `exit` e in cima al *Data Stack* viene inserito l'indirizzo di memoria della *struct* `netcon` associata al *socket* del client.
- `netcon-send-buf ( netcon buffer len -- | throws:ENETCON )`: scrive su un *socket* UDP `netcon` il contenuto di un *buffer* `buffer` di lunghezza `len`. Questa operazione è eseguita usando la parola `netcon-send` definita in *Assembly* che richiama la funzione corrispondente dell'SDK consumando i parametri sopra specificati; questa funzione ritorna un codice di stato che viene analizzato dalla parola `check`, lanciando l'eccezione `ENETCON` in caso di errore.
- `netcon-write-buf ( netcon buffer len -- | throws:ENETCON )`: questa parola è come la `netcon-send-buf` ma è usata nel caso di *socket* TCP. Utilizza la parola `netcon-write` definita in *Assembly* che ha lo stesso comportamento della `netcon-send` discussa precedentemente.
- `netcon-write ( netcon str -- | throws:ENETCON )`: ridefinita tramite *override* rispetto a quella definita in *Assembly* discussa

precedentemente, questa parola ha il compito di scrivere su un *socket* TCP `netcon` una stringa.

- `netcon-writeln ( netcon str -- | throws:ENETCON )`: questa parola ha lo stesso comportamento della `netcon-write`, ma scrive in più i caratteri *CR LF* sul *socket*.
- `read-ungreedy ( size buffer netcon -- count code | throws: ERTIMEOUT )`: questa parola si occupa di leggere dal *socket* `netcon` un numero `size` di byte, che scriverà all'interno del *buffer* `buffer`. Al suo interno è utilizzata la parola `netcon-recvinto`, definita in *Assembly*, che richiama la funzione corrispondente dell'SDK; questa funzione utilizza i parametri sopra indicati per leggere i `size` byte direttamente dal *socket*, restituendo la `struct recvinto_res` che conterrà al suo interno un codice di stato e il numero (`count`) di byte letti. Questi due valori sono posizionati in cima al *Data Stack* dalla parola `netcon-recvinto`. La parola `read-ungreedy` ha al suo interno un ciclo `begin again` che si occupa di effettuare l'operazione di lettura, esaminando il codice di stato restituito: se si sono verificati *timeout* del tipo `NC_ERR_TIMEOUT`, il ciclo continua richiamando la `netcon-recvinto`, altrimenti si effettua l'interruzione del ciclo (avendo eseguito prima la pulizia del *Return Stack* utilizzato) terminando così l'esecuzione della parola e lasciando, in cima al *Data Stack*, i valori `count` e `code`. La parola `read-ungreedy` tiene conto anche di un *timeout* proprio che viene valutato attraverso i valori ottenuti tramite la parola `ms@`; all'interno della ESP8266 non è presente un orologio di sistema costantemente alimentato che tenga conto dello scorrere del tempo ma, con la parola `ms@`, definita in *Assembly*, è possibile ottenere un'informazione simile: infatti, questa parola richiama una funzione presente all'interno dell'SDK che fornisce il tempo di esecuzione (in millisecondi) da quando lo *scheduler* del sistema operativo è stato avviato; quindi (tornando alla parola `read-ungreedy`) se la differenza tra il valore corrente di `ms@` e il valore ottenuto all'inizio dell'esecuzione della parola supera il *timeout* impostato per il *socket* da cui si sta leggendo, si lancia l'eccezione `ERTIMEOUT` che interrompe il ciclo.
- `netcon-read ( netcon size buffer -- count | -1 | throws:ENETCON / ERTIMEOUT )`: questa parola legge dal *socket* `netcon` un numero massimo `size` di byte e li scrive all'interno del *buffer* `buffer` usando la parola `read-ungreedy`. In caso di errore, si lanciano le eccezioni

`ENETCON` e `ERTIMEOUT` in base al problema rilevato. Se non ci sono stati errori, la parola rilascia in cima al *Data Stack* il numero `count` di byte letti. Se la connessione è stata chiusa, la parola `read-ungreedy` rilascia in cima al *Data Stack* il valore corrispondente alla costante `NC_ERR_CLSD`: in questo caso la parola `netcon-read` rilascia in cima al *Data Stack* il valore `-1` per segnalare la chiusura del *socket*.

Durante l'attività progettuale, questa parola non ha funzionato correttamente: dalla sua seconda esecuzione in poi (per motivi che non sono si sono individuati) il risultato restituito è sempre `0`. Per risolvere questo problema, si è provveduto a modificare definizione di questa parola.

- `netcon-readln ( netcon size buffer -- count | -1 | throws:ENETCON / EOVERFLOW / ERTIMEOUT )`: questa parola legge una linea dal *socket* `netcon` il cui terminatore è dato dai caratteri *CR LF*, al massimo per una dimensione `size`. I caratteri letti vengono scritti all'interno del *buffer* `buffer`. In caso di errore, si lanciano le eccezioni `ENETCON`, `EOVERFLOW` e `ERTIMEOUT` in base al problema rilevato. L'eccezione `EOVERFLOW` viene lanciata se il *buffer* non è sufficientemente grande. Se non ci sono stati errori, la parola rilascia in cima al *Data Stack* il numero `count` di byte letti o `-1` se la connessione è stata chiusa. Questa parola esegue un ciclo `do loop` usando la parola `netcon-read` per leggere dal *socket* un carattere alla volta; il carattere letto viene memorizzato nella posizione i-esima di `buffer`. La lettura di un carattere alla volta viene eseguita per controllare la presenza della sequenza *CR LF* in modo da interrompere la lettura alla loro occorrenza.
- `netcon-dispose ( netcon -- )`: questa parola chiude e rende disponibile un *socket* `netcon` aperto usando le parole `netcon-close` e `netcon-delete` definite in *Assembly* che utilizzano le corrispondenti funzioni dell'SDK.

Si è voluto prestare particolare attenzione a queste parole perché, tra le parole direttamente definite dallo sviluppatore di Punyforth per la ESP8266, sono state quelle più utilizzate all'interno dell'attività progettuale svolta. Queste, infatti, sono le parole che consentono la comunicazione a livello di rete e che hanno dato la possibilità (come si vedrà nel capitolo 6) di scrivere client, server e diversi protocolli applicativi, semplicemente usando del codice simbolico, senza dovere ricorrere

direttamente a linguaggi di livello più basso. Si riporta di seguito l'esempio di scrittura un semplice server e client UDP.

```
\ UDP Server
128 buffer: data

8000 "172.26.93.2" netcon-udp-server
dup 128 data netcon-readln
print: 'received bytes: ' . cr
data type
netcon-dispose

\ UDP Client
8000 "172.26.93.2" UDP netcon-connect
dup "LAMP ON" 7 netcon-send-buf
netcon-dispose
```

4.7.4 Task e Mailbox

Altro elemento fondamentale presente in Punyforth è il *multitasking*: è possibile eseguire più *task* contemporaneamente facendoli cooperare e comunicare tra loro. In questo modo, ad esempio, è possibile avviare un *task* per il server che fornisce un particolare servizio e diversi *task* per servire i client che man mano si connettono; oppure, ad esempio, si potrebbe avviare un client *MQTT* e contemporaneamente avere l'accesso remoto alla *shell*, per potere inviare e ricevere codice simbolico, attraverso un server *REPL* avviato al suo interno. Quelli proposti, sono diversi scenari possibili, praticamente realizzati e funzionanti.

L'ambiente simbolico offre una reale implementazione dei task mediante opportune parole che si trovano all'interno del modulo *tasks.forth* (consultabile alla sezione 9.1.3). L'implementazione effettuata prevede l'inizializzazione di un *task* tramite la parola `task:`.

Questa parola crea, per ogni *task*, una *struct* `Task` che contiene i parametri che definiscono l'area di memoria del *task* che si sta creando e rappresenta un nodo di una lista concatenata circolare che collega i vari *task*. La parola `task:` inizializza i vari campi di questa *struct* nel seguente modo:

1. il campo `.status` è impostato a `SKIPPED` (l'atro possibile stato è `PAUSED`);

2. il campo `.next` è inizializzato con il valore del campo `.next` del *task* che lo precede (l'indirizzo di memoria del *task* precedente è contenuto all'interno della variabile `last`);
3. il campo `.next` del *task* precedente è aggiornato con l'indirizzo di memoria del *task* corrente;
4. si allocano gli *stack pointer* del *Data Stack* e del *Return Stack*, le cui basi saranno rispettivamente memorizzate nei campi `.sp` e `.rp`, della dimensione presente all'interno della variabili `task-stack-size` e `task-rstack-size`;
5. si inizializza l'eccezione corrente a `0` memorizzando questo valore nel campo `.handler`;
6. si memorizzano i *TOS* del *Data Stack* e del *Return Stack* rispettivamente nei campi `.s0` e `.r0`;
7. si aggiorna la variabile `last` con l'indirizzo di memoria del *task* appena creato.

Un *task* si avvia tramite la parola `activate` che ha il compito di settare a `PAUSED` lo stato del *task* corrente, memorizzando i valori attuali delle basi di `rp`, `ip` e `sp` all'interno dei campi `.rp`, `.ip` e `.sp` tramite la parola `save`; successivamente si esegue lo `switch` tra il *task* precedente e quello corrente che imposta all'interno della variabile `current` l'indirizzo della *struct* del *task* che si sta attivando ed esegue il `restore` degli *stack pointer* `sp` e `rp` e dell'*instruction pointer* `ip` che erano stati memorizzati all'interno dei campi `.sp`, `.rp` e `.ip`.

Un *task* si interrompe tramite la parola `deactivate` che ha il compito di settare lo stato del *task* corrente a `SKIPPED`, per poi eseguire la parola `choose` che, eseguendo un ciclo `begin until` per esplorare la lista dei *task* usando il campo `.next` di ogni nodo, si occupa di selezionare il primo *task* disponibile che ha il campo `.status` impostato con il valore `PAUSED`; sul *task* individuato si eseguirà la parola `switch` per cambiare su di esso il contesto di esecuzione.

Per usare più *task*, bisogna prima cambiare la modalità da `single-task` a `multi-task`: questa operazione si esegue attraverso la parola `multi`.

I *task* possono comunicare tra loro usando le *mailbox*; ogni *mailbox* è una coda con blocchi di dimensione fissa dove dei messaggi possono essere lasciati per un *task*.

Le parole delle *mailbox* sono state implementate all'interno del modulo *mailbox.forth* (consultabile alla sezione 9.1.4).

Una *mailbox* si crea tramite la parola `mailbox` che usa la parola `ringbuf: ( capacity ) ( -- ringbuffer )` definita all'interno del modulo *ringbuf.forth*; questa parola si occupa di creare la `struct RingBuf` che rappresenta un *buffer* circolare.

Un messaggio si inserisce all'interno di una *mailbox* tramite la parola `mailbox-send` dove si esegue un ciclo `begin while repeat` che controlla che la *mailbox* `mailbox` sia piena: in questo caso, infatti, il ciclo continua interrompendo ripetutamente il *task* corrente (tramite la parola `pause`) finché almeno uno slot all'interno del *buffer* non si sia liberato (perché qualche altro *task*, con accesso a quella particolare *mailbox*, ha consumato un messaggio); se il ciclo viene interrotto, perché all'interno della *mailbox* vi è almeno uno slot libero, si esegue la parola `ringbuf-enqueue` che memorizza l'indirizzo di memoria del messaggio `message` al suo interno.

Il procedimento inverso viene eseguito dalla parola `mailbox-receive` che si occupa di ricevere i messaggi. Questa parola esegue la stessa tipologia del ciclo precedente controllando che il *buffer* `mailbox` sia vuoto: in questo caso, infatti, il ciclo continua interrompendo il *task* corrente ogni volta che non ci sono messaggi all'interno della *mailbox*; se invece c'è almeno un messaggio all'interno della *mailbox*, il ciclo si interrompe e si esegue la parola `ringbuf-dequeue` che elimina il messaggio `message` dal *buffer* e mette il suo indirizzo di memoria in cima al *Data Stack*.

Di seguito si presenta un esempio di *task* (`task-consumer`) che si occupa di ricevere e stampare (sull'*output stream*) i messaggi contenuti in una *mailbox* (`mailbox1`, che può contenere fino a 5 messaggi) inviati dal *task* principale dell'interprete. Il *task* esegue la parola `consumer` che si occupa di eseguire le operazioni di ricezione e stampa interrompendo il *task* secondo il funzionamento della parola `mailbox-receive` illustrata in precedenza.

```
5 mailbox: mailbox1
task: task-consumer
: consumer ( task -- )
    activate
    begin
    mailbox1 mailbox-receive .
```

```
        println: "received by consumer"
        pause
        again
        deactivate ;
    multi
    task-consumer consumer
    123 mailbox1 mailbox-send
    456 mailbox1 mailbox-send
```

4.7.5 Flash

Punyforth si è evoluto diverse volte durante l'attività progettuale e, nell'ultima versione su cui si è lavorato, è stato aggiunto un modulo, il *flash.forth*, che ha aperto la strada verso nuove implementazioni. Questo modulo contiene le definizioni che consentono al programmatore di poter eseguire le operazioni di lettura, scrittura e cancellazione sulla flash SPI della ESP8266. Con questo modulo, quindi, è possibile mantenere memorizzate e ottenere informazioni anche dopo il riavvio del dispositivo; una grande potenzialità di questo aspetto è la possibilità di memorizzare direttamente del codice simbolico all'interno dei blocchi di flash (come ad esempio una configurazione, oppure delle parole che eseguono delle operazioni specifiche per quel particolare nodo) caricando all'avvio di Punyforth i blocchi di flash interessati e passando all'interprete il codice simbolico per poterlo compilare, per poi inserire le definizioni all'interno del suo dizionario, il tutto senza dovere utilizzare l'unità UART per caricare all'interno della flash SPI queste informazioni perché possono essere caricate anche attraverso l'accesso remoto, inviando le stringhe di codice simbolico da scrivere e le parole per la scrittura in flash (sezione 6.6).

La flash SPI è suddivisa in blocchi da 4096 byte ciascuno; nel caso della ESP8266-12E utilizzata per l'attività progettuale, la flash SPI è grande 4 MB. Punyforth mette a disposizione del programmatore le seguenti parole per interagire con la flash SPI:

- `erase-flash (block -- status_code):` questa parola è stata definita in *Assembly* e usa la funzione corrispondente dell'SDK; consuma dal *Data Stack* l'intero `block` che indica il numero del blocco di flash (1 = primo blocco, da 0 a 4095 byte; 2 = secondo blocco, da 4096 a 8191; …); questa parola eseguirà la cancellazione dell'intero blocco specificato rilasciando in cima al *Data Stack* un codice di stato (`status_code`) dell'operazione

eseguita (`0`: `SPI_FLASH_RESULT_OK`; `1`: `SPI_FLASH_RESULT_ERR`; `2`: `SPI_FLASH_RESULT_TIMEOUT; 3`: per i casi non previsti).

- `write-flash (size buffer address -- status_code)`: questa parola è stata definita in *Assembly* e usa la funzione corrispondente dell'SDK; scrive sulla flash le informazioni contenute all'interno di `buffer` di dimensione `size` a partire dall'indirizzo `address` (che non è espresso in blocchi, ma in byte; se si volesse fare la conversione da blocchi a byte, si può usare la parola `block`). Al termine, rilascia in cima al *Data Stack* un codice di stato (`status_code`) come la parola precedente.
- `load (block -- )`: questa parola è stata definita in *Assembly* e usa delle funzioni scritte in C dallo sviluppatore di Punyforth (il sorgente è all'interno del file *forth_io.c*); il suo compito è quello di leggere i dati presenti a partire da un blocco `block` di *flash* SPI passando i caratteri letti all'interprete e interrompendo l'operazione non appena incontra il *token* di chiusura `\n\n/end\n\n`.
- `list (block -- )`: questa parola, definita all'interno del modulo *flash.forth*, ha il compito di leggere l'intero blocco carattere per carattere, stampando sull'*output stream* i caratteri letti.

Durante l'attività progettuale, utilizzando queste parole, si è rilevato che non è possibile scrivere un blocco di flash se prima non lo si cancella tramite la parola `erase-flash`: in questo caso la scrittura non va a buon fine e, al successivo comando che si proverà ad eseguire, la ESP8266 eseguirà un riavvio. Questo fa capire che, se si volesse utilizzare la memoria flash SPI come dispositivo di memorizzazione temporaneo, ad esempio per sopperire alla carenza di memoria disponibile in RAM, si dovrebbero gestire ripetute operazioni di salvataggio, cancellazione e ripristino dei dati contenuti tutte le volte che si volesse aggiungere qualcosa ad un blocco.

Di seguito un esempio:

```
256 erase-flash
SIZE ": test 5 5 + . ;\n\n/end\n\n" SIZE 256 * write-flash
256 load
```

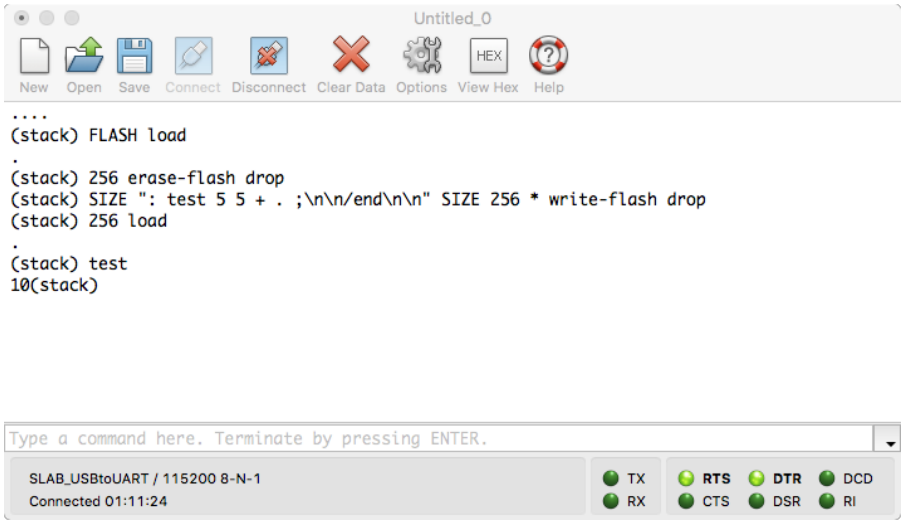

Figura 10 - Memorizzazione e lettura di codice simbolico nella flash SPI

Il codice dell'esempio (la cui esecuzione è visibile in Figura 10) si occupa di cancellare il blocco `256`, di memorizzare al suo interno (tramite la parola `write-flash`) il codice simbolico che definisce la parola `test` (che effettua la somma tra due numeri e ne stampa il risultato sull'*output stream*) e infine di leggere il contenuto della flash SPI (tramite la parola `load`, a partire dall'indirizzo del blocco `256`); a questo punto si passa il codice simbolico all'interprete che esegue il codice per effettuare la *Colon Definition* della parola `test`. Come si può vedere dalla Figura 10, l'esecuzione della parola `test` dalla *shell* di Punyforth avviene con successo (perché la voce è stata trovata all'interno del dizionario) mostrando il risultato `10` della somma dei due numeri `5`. Come si può vedere dalla Figura 11, dopo il riavvio della ESP8266-12E è stato possibile caricare, tramite la parola `load`, il contenuto del blocco `256` che è stato interpretato regolarmente dall'interprete di Punyforth.

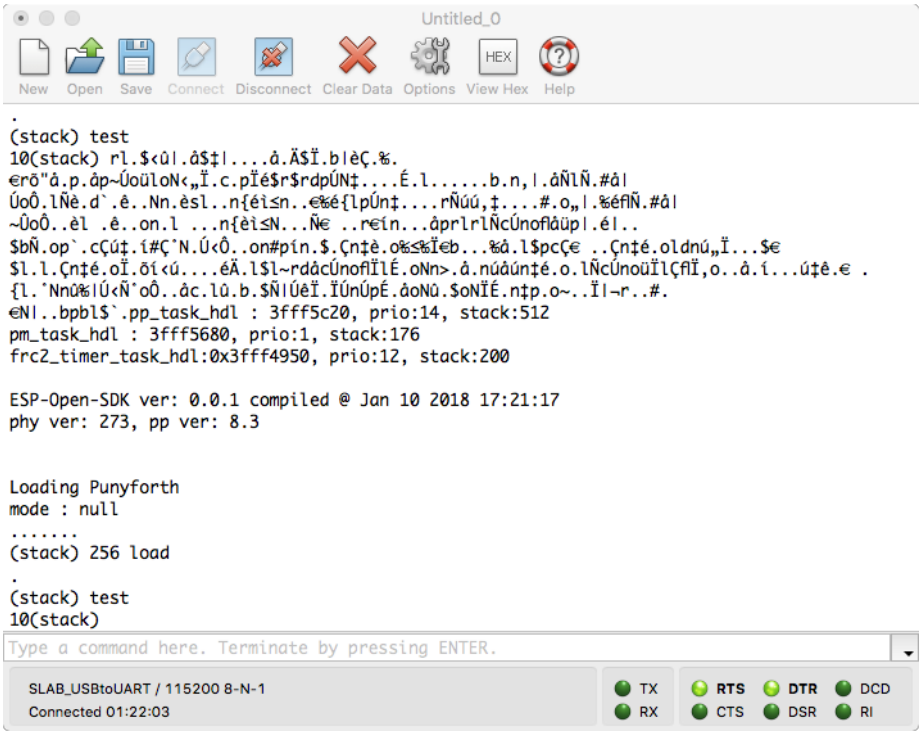

Figura 11 - Caricamento del contenuto del blocco di flash SPI dopo il riavvio

4.7.6 TCP REPL

Il modulo *tcp-repl.forth* contiene un altro elemento importantissimo per il contesto dell'IoT. Infatti, attraverso questo modulo, è possibile collegarsi ad una *shell* remota che consente all'utente di inviare (ad esempio attraverso un terminale Telnet e attraverso una rete TCP/IP) delle stringhe contenente codice simbolico che saranno interpretate ed eseguite. Il TCP *REPL* utilizza i moduli *netcon.forth*, *task.forth*, *wifi.forth* e *mailbox.forth* precedentemente illustrati e rappresenta un esempio pratico del loro utilizzo.

Il modulo contiene: le costanti `HOST` e `PORT` che rappresentano rispettivamente l'interfaccia e il numero di porta assegnati al server, la variabile `client` che contiene l'indirizzo di memoria del *socket* `netcon` associato all'ultimo client connesso, il *buffer* `line` che serve per contenere i caratteri ricevuti dal client (128 caratteri al massimo), i *task* `repl-server-task` e `repl-worker-task` che servono per contenere rispettivamente i *task* del server e del *worker* (usato per gestire le richieste del client) e la *mailbox* `connections` che serve per comunicare l'arrivo di un nuovo client al *task worker*.

Al suo interno si trovano definite le seguenti parole:

- `server (task -- )`: questa parola contiene le istruzioni del *task* che si occupa di eseguire le operazioni del server. Il suo compito, quindi, è quello di

aprire ed effettuare il *binding* di una porta TCP usando la parola `netcon-tcp-server` e di avviare un ciclo `begin again` che si occupa di attendere un client tramite la parola `netcon-accept`; non appena un client effettuerà la connessione, il suo *socket* `netcon` sarà inviato tramite mailbox ad un *task* `worker` che si occuperà di gestire le sue richieste.

- `command-loop ( -- )`: questa parola contiene un ciclo `begin while repeat` che si occupa di leggere i caratteri inviati dal client sul *socket* `netcon client` (tramite la parola `netcon-readln`); questi caratteri vengono memorizzati all'interno del *buffer* `line`. Il suo contenuto è poi interpretato dalla parola `eval`: se `line` contiene una stringa di codice simbolico valido, questa viene eseguita. Il ciclo continua finché il *socket* `netcon` del client sarà aperto (quindi finché il valore restituito dalla `netcon-readln` sarà diverso da -1) e finché il contenuto del *buffer* `line` sarà diverso dalla parola `quit`.
- `worker (task -- )`: questa parola contiene le istruzioni del *task* che si occupa di gestire le richieste del client. Al suo interno è presente un ciclo `begin again` che, inizialmente, rimane in attesa che il *task* del server mandi (tramite *mailbox*) l'indirizzo di un *socket* `netcon` di un nuovo client. Non appena la *mailbox* `connections` conterrà il nuovo client, il *worker* memorizzerà l'indirizzo del *socket* `netcon` del client all'interno della variabile `client` ed eseguirà la parola `command-loop`.
- `eval (str -- )`: questa parola ha il compito di leggere il contenuto del *buffer* `str`, carattere per carattere, passando ogni carattere letto all'*input buffer* (su cui lavora l'interprete di Punyforth); dopo avere caricato tutti i caratteri del *buffer* `str`, la parola `eval` carica all'interno dell'*input buffer* anche i caratteri *CR LF* ed esegue la parola `push-enter` per dare luogo all'interpretazione dei caratteri caricati.
- `repl-start ( -- )`: questa parola avvia il server TCP REPL. Oltre ad impostare la modalità *multi-task*, modifica anche il comportamento di esecuzione delle parole `type` ed `emit` assegnando rispettivamente il comportamento delle parole `type-composite` ed `emit-composite`; queste parole eseguono le seguenti operazioni: se la variabile `client` contiene un *socket* `netcon` client valido, inviano l'*output* al client tramite la `netcon-write`, altrimenti inviano l'*output* all'*output stream*. Terminate queste operazioni, la parola `repl-start` avvia i *task* `server` e `worker`.

Questo modulo è stato molto utile all'interno dell'attività progettuale svolta perché ha consentito di implementare all'interno del sistema di regole (come si vedrà nel capitolo 6) tutta la parte dedicata alla comunicazione delle stringhe di codice simbolico; queste stringhe, generate automaticamente dal sistema di regole, riguardano la configurazione e l'esecuzione di operazioni sui diversi nodi della rete. Inoltre, questo modulo è stato fonte di ispirazione per l'implementazione, in codice FORTH, di altri aspetti riguardanti alcuni protocolli utilizzati nel contesto dell'IoT.

5 Ambiente di sviluppo

Com'è stato detto all'inizio del capitolo 4, Punyforth supporta principalmente il microcontrollore Wi-Fi ESP8266. Per questa piattaforma l'immagine binaria di base contiene diverse parole attraverso le quali, direttamente dalla *shell* dell'interprete di Punyforth, è possibile eseguire diverse operazioni per potere interagire con tutti gli aspetti e le funzionalità che questo microcontrollore mette a disposizione; inoltre, la modalità *open source* con cui il progetto di Punyforth è stato distribuito dal suo sviluppatore e le possibilità di estensione del suo dizionario, hanno consentito la modifica e l'implementazione di diverse altre parole che sono state utili per l'attività progettuale svolta.

In questo capitolo si farà un'analisi della struttura di Punyforth rispetto all'architettura della ESP8266, mostrando tutti i passi necessari per la compilazione dell'immagine binaria e per la sua installazione.

5.1 Scheda di sviluppo

Per l'attività progettuale svolta, è stata usata la ESP8266-12E installata all'interno di una scheda di sviluppo *NodeMCU Amica*. Su questa scheda è presente il *chip CP2102* della *Silicon Labs*, cioè un *bridge* USB to UART che consente l'accesso seriale alla ESP8266 utilizzando la sua interfaccia UART (di cui è stato discusso nella sezione 3.3.5) tramite la porta micro USB presente sulla scheda di sviluppo. Inoltre, questa scheda mette a disposizione dello sviluppatore 15 pin *header* presenti su 2 lati, attraverso i quali è possibile accedere alle GPIO, ai pin di alimentazione e ai pin con funzioni dedicate discussi nella sezione 3.3. Sono presenti anche 2 pulsanti, *RST* per il reset della scheda e *FLASH* per abilitare la modalità di download su alcuni sistemi operativi Linux.

Non vi è una corrispondenza diretta tra i pin della NodeMCU Amica e quelli della ESP8266-12E: si può notare da subito la differenza in numerosità: 30 pin *header* per la NodeMCU Amica contro i 22 pin per la EPS8266-12E. Infatti, sulla NodeMCU Amica sono presenti 6 pin *header* aggiuntivi per l'alimentazione (3.3V e GND) e 2 pin *header* ad uso riservato. La mappatura tra i pin della ESP8266 e i pin *header* della scheda di sviluppo NodeMCU Amica sono stati riportati in Tabella 4 e in Figura 12.

NodeMCU Amica	ESP8266-12E		
	Function 1	Function 2	Function 3
D0	GPIO16		
D1	GPIO5		
D2	GPIO4		
D3	GPIO0	FLASH	
D4	GPIO2	TXD1	
D5	GPIO14	HSCLK	
D6	GPIO12	HMISO	
D7	GPIO13	RXD2	HMOSI
D8	GPIO15	TXD2	HCS
RX	GPIO3	RXD0	
TX	GPIO1	TXD0	
Vin	VCC		
GND	GND		
RST	RST		
EN	EN		
CLK	SCLK	SDCLK	
SD0	MISO	SDD0	
CMD	CS	SDCMD	
SD1	MOSI	SDD1	
SD2	GPIO9	SDD2	
SD3	GPIO10	SDD3	
A0	ADC0	TOUT	

Tabella 4 - Mappatura Pin NodeMCU Amica e ESP8266-12E

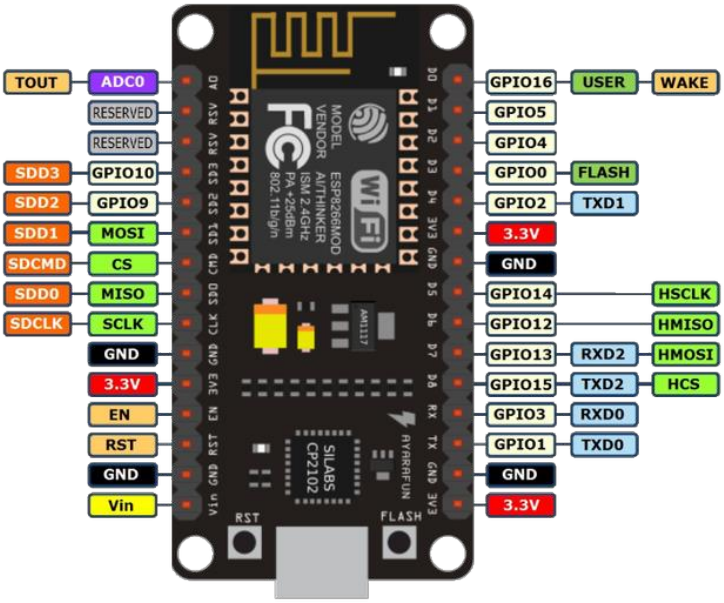

Figura 12 - NodeMCU Amica pin header

5.2 Download e installazione

Il progetto di Punyforth è disponibile su un repository *GitHub* [17] da cui è possibile scaricare il file zip del progetto; se si dispone un client *git* installato sulla propria macchina, è anche possibile effettuare il *clone* del progetto tramite il comando

```
git clone https://github.com/zeroflag/punyforth.git
```

il quale scaricherà l'intero progetto su una cartella locale. Il commit a cui attualmente si fa riferimento, e su cui si è basata l'intera attività progettuale svolta, è il *653ef3c05c2efe635a079647326b0d8b3e9d25dd* del 12 Maggio 2018.

Se si volesse installare immediatamente Punyforth all'interno della ESP8266, sarebbe possibile farlo fin da subito senza dover necessariamente compilare i sorgenti per ottenere l'immagine binaria. Infatti, all'interno della cartella *arch/esp8266/bin*, è già presente l'immagine binaria compilata *punyforth.bin*. All'interno della stessa cartella sono presenti gli strumenti per effettuare il download dei file binari e dei moduli all'interno della memoria flash SPI della ESP8266: per eseguire questa operazione si usa lo script Python *flash.py* che utilizza lo script Python *esptool.py* [18], uno strumento per comunicare con il *bootloader* delle ESP8266 e delle ESP32. Tramite il comando

```
python flash.py <serial_port>
```

si esegue lo script *flash.py*, dove `<serial_port>` è la porta seriale a cui è collegata la ESP8266 che potrebbe cambiare in base al sistema operativo in uso; si veda la Tabella 5 per alcuni esempi di come la porta seriale potrebbe risultare in base al sistema operativo.

Sistema Operativo	Porte
Linux	/dev/ttyS* o /dev/ttySUSB*
Mac OSX	/dev/cu.SLAB_USBtoUART*
Windows	COM*

Tabella 5 - Porte Seriali per Sistemi Operativi

Nello specifico, attraverso il commando precedente, si effettuerà il download dei file *rboot.bin*, *punyforth.bin* e di tutti i moduli specificati all'interno della lista `AVAILABLE_MODULES`, accessibile dallo scope globale, all'interno del file *flash.py*.

Per l'utilizzo del CP2102 su sistemi operativi Windows e Mac OSX, è stato necessario installare i driver reperibili direttamente dal sito del produttore. Si rende noto anche che al termine della procedura di flash, la ESP8266 non si riavviava, per cui si è reso necessario il riavvio manuale attraverso l'apposito segnale di reset.

5.3 Tools

Come si è potuto vedere dalla sezione precedente, vi sono diversi strumenti che consentono di eseguire diverse operazioni sulla ESP8266. In questa sezione saranno illustrati quelli che sono stati utilizzati durante l'attività progettuale.

5.3.1 flash.py

Lo script *flash.py* è uno strumento molto utile prodotto dallo sviluppatore di Punyforth per gestire il caricamento dei file binari e dei moduli all'interno della flash SPI della ESP8266.

Nella sua forma completa di utilizzo si presenta in questo modo

```
python flash.py <port> --modules <value> --binary <value> --
  main <value> --flashmode <value> --block-format <value>
```

In Tabella 6 è presente l'elenco dei parametri e delle opzioni con la loro descrizione e i loro valori predefiniti.

Parametro / Opzione	Descrizione	Predefinito
<port>	Porta seriale della ESP8266. Questo parametro è obbligatorio.	
--modules	Lista dei moduli separati dal simbolo spazio.	all
--binary	Valore booleano. Se impostato a false, i file binari (rboot.bin e punyforth.bin) non saranno caricati all'interno della flash SPI. Se impostato a true avrà il comportamento opposto.	true
--main	Percorso del file contenente il codice Forth che sarà usato come script iniziale all'avvio di Punyforth.	
--flashmode	Definisce la modalità flash utilizzata. I valori validi sono qio, qout, dio, dout.	qio
--block-format	Valore booleano. Se impostato a true, i sorgenti Forth saranno formattati secondo il formato a blocchi di 128 caratteri per riga e 32 righe, usando il carattere di spazio come padding.	false

Tabella 6 - Parametri flash.py

Durante l'attività progettuale si sono utilizzate le seguenti opzioni:

- `--modules`: ha consentito di specificare i soli moduli necessari, in modo da risparmiare spazio all'interno della flash SPI e all'interno del dizionario.
- `--main`: ha consentito di caricare il sorgente contenente il codice simbolico da eseguire all'avvio di Punyforth.
- `--binary`: impostato a `false`, in fase di sviluppo dei sorgenti contenenti codice simbolico, ha consentito di effettuare il caricamento dei soli moduli, risparmiando il tempo di flash dei file binari.
- `--flashmode`: si è dovuta impostare la modalità flash `dio` perché con la modalità predefinita (`qio`) la procedura di flash non caricava correttamente i dati.

5.3.2 esptool.py

Lo script Python *esptool.py* [18] è uno strumento *open source*, indipendente dal sistema operativo di utilizzo, che consente di interfacciarsi con il *bootloader* presente all'interno della ROM delle ESP8266 e delle ESP32. Attraverso questo

strumento, è possibile eseguire diverse operazioni e si usa attraverso il seguente comando

```
python esptool.py -p <port> -b <baudrate> <command>
```

Attraverso l'opzione `-p` (o `--port`) si specifica la porta seriale a cui è collegata l'MCU; tramite la variabile d'ambiente `ESPTOOL_PORT` è possibile specificare la porta seriale da utilizzare come predefinita. Se l'opzione `-p` o la variabile `ESPTOOL_PORT` non sono settati, lo script provvederà ad esplorare tutte le porte seriali connesse provandole una alla volta finché non troverà un dispositivo *Espressif* collegato.

Attraverso l'opzione `-b` (o `--baud`) si può specificare il *baudrate* se diverso da quello di default che è impostato a 115200bps. È possibile usare la variabile d'ambiente `ESPTOOL_BAUD` per specificare il *baudrate* da usare come predefinito.

Attraverso l'opzione `-h` è possibile ottenere la lista dei comandi che possono essere eseguiti; tramite

```
python esptool.py <command> -h
```

è possibile ottenere la lista dei parametri da utilizzare per i vari comandi. Di seguito si farà l'elenco dei comandi che si sono rilevati più utili.

5.3.2.1 Comandi per la flash SPI

Attraverso questi comandi è possibile scrivere, leggere e cancellare, tramite l'interfaccia seriale, dati binari all'interno della flash SPI.

Il comando `write_flash` si occupa di scrivere dati binari all'interno della flash

```
python esptool.py -p <port> write_flash <address> <file_name>
```

dove: il parametro `<address>` indica in formato esadecimale o decimale l'indirizzo da cui iniziare l'operazione di flash; il parametro `<file_name>` indica il file da scrivere. È possibile scrivere più di un file alla volta specificando più coppie `<address> <file_name>` di seguito. Questo comando consente anche di specificare la modalità di flash tramite l'opzione `--flash_mode` e la dimensione della flash SPI tramite l'opzione `--flash_size`.

Per quanto riguarda le modalità di accesso alla flash SPI, le ESP8266 e le ESP32 supportano due tipologie di modalità: *Quad SPI* e *Dual SPI*. Queste tipologie

di modalità raggruppano rispettivamente le modalità *qio* - *qout* e *dio* - *dout* (vedi Tabella 7).

Tipologia	Modalità	Nome	Pin usati	Velocità
Quad SPI	qio	Quad I/O	4 pin usati per indirizzi e dati	La modalità più veloce
	qout	Quad Output	4 pin usati per i dati	Cira il 15% più lenta della qio
Dual SPI	dio	Dual I/O	2 pin usati per indirizzi e dati	Cira il 45% più lenta della qio
	dout	Dual Output	2 pin usati per i dati	Cira il 50% più lenta della qio

Tabella 7 - Modalità di accesso alla flash SPI per ESP8266 e ESP32

Di seguito un esempio d'uso del comando `write_flash`

```
python esptool.py -p /dev/ttyUSB1 write_flash --flash_mode
dio --flash_size 4MB 0x0 bootloader.bin
```

Il comando `read_flash` si occupa di leggere il contenuto della flash SPI

```
python esptool.py -p <port> read_flash <address> <size>
<filename>
```

dove `<address>` è l'indirizzo da cui fare partire la lettura, `<size>` è la dimensione di flash da leggere (a partire da `<address>`) e `<filename>` è il file di *output*. È possibile aggiungere l'opzione `--progress` per mostrare l'andamento della lettura.

I comandi `erase_flash` e `erase_region` consentono la cancellazione dell'intera flash o soltanto alcune sue porzioni

```
python esptool.py -p <port> erase_flash
```

```
python esptool.py -p <port> erase_region <address> <size>
```

5.3.2.2 Comandi per ROM e RAM

Attraverso questi comandi, usando sempre l'interfaccia seriale, è possibile leggere e scrivere direttamente sullo spazio di memoria del *chip*.

Il comando `dump_mem` consente di effettuare il *dump* di una porzione di memoria

```
python esptool.py -p <port> dump_mem <address> <size>
                    <filename>
```

Il commando `load_ram` consente di caricare un'immagine binaria eseguibile direttamente in RAM per poi eseguire il programma contenuto al suo interno al termine del caricamento.

```
python esptool.py -p <port> load_ram <filename>
```

dove `<filename>` è il file dell'immagine binaria che deve contenere soltanto segmenti di IRAM o DRAM.

Il comando `write_mem` è simile al precedente, solo che consente di caricare in RAM una sola parola (di 4 byte) `<value>` nell'indirizzo `<address>`

```
python esptool.py -p <port> write_mem <address> <value>
```

Il commando `read_mem`, invece, consente di leggere una parola in RAM all'indirizzo `<address>`

```
python esptool.py -p <port> <address>
```

5.4 Struttura di Punyforth

Punyforth non è un sistema operativo, ma un'applicazione che gira come *task* all'interno di una versione dedicata alla ESP8266 di *FreeRTOS*. Come anticipato in precedenza, il suo codice sorgente è fatto da diversi file scritti in linguaggio C (che usano le funzioni di *FreeRTOS* e dell'*OpenSDK*), diversi file scritti in linguaggio *Assembly* (attraverso i quali lo sviluppatore ha implementato l'interprete FORTH) e diversi file scritti in linguaggio FORTH dove sono state implementate (direttamente usando il codice simbolico interpretato) la maggior parte delle parole trattate nel capitolo 4.

All'interno della cartella *punyforth* possiamo distinguere le cartelle *arch* e *generic* che contengono rispettivamente i file dedicati alle diverse architetture supportate (ARM, x86 e ESP8266) e i file comuni tra le diverse architetture.

Cominciamo dalla cartella *generic*. Questa cartella contiene file scritti in linguaggio FORTH (contenuti a loro volta all'interno della cartella *forth*) e file scritti

in linguaggio *Assembly*. Tra i file scritti in linguaggio *Assembly*, si è scelto di descrivere i file che si sono utilizzati direttamente durante l'attività progettuale:

- *data.S*: definisce la sezione `.data`; al suo interno troviamo definite le aree di memoria dedicate agli *stack* (*Data Stack* e *Return Stack*), all'*input buffer*, al dizionario, all'*heap* e a diverse costanti. Durante l'attività progettuale, questo file è stato modificato per incrementare lo spazio riservato all'*heap*.
- *macros.S*: definisce le macro `defprimitive` e `defword` utilizzate per definire nuove parole all'interno del dizionario direttamente da linguaggio *Assembly*.
- *words.S*: contiene alcune parole definite direttamente in linguaggio *Assembly*.

Il file *outerinterpreter.S* contiene le direttive per l'interprete interno: cerca ed esegue le parole all'interno del dizionario saltando alla *label* contenente la procedura di errore nel caso in cui la parola non sia nel dizionario; in questo caso, valuta se la stringa inserita è un numero: se non è un numero, salta alla *label* dedicata alla gestione di questo errore.

All'interno della cartella *arch* si trova la cartella `esp8266` contenente i file dedicati per l'architettura della ESP8266. Al suo interno si trovano le cartelle: *bin* (contenente le immagini binarie, il file *main.forth* del programma che viene eseguito all'avvio di Punyforth e i *tools* per il caricamento di questi file in flash), *forth* (contenente i moduli FORTH esaminati nel capitolo precedente) e *rtos/user* (contenente i sorgenti scritti in linguaggio C).

All'interno della cartella *rtos/user*, durante l'attività progettuale, si sono rilevati utili i seguenti file:

- *punyforth.S*: all'interno di questo file (oltre ad essere definita la porzione di codice che si occupa di inizializzare i registri per avviare l'interprete di Punyforth) è possibile aggiungere i propri file *.S* utilizzando la direttiva `#include`. Se questi file definiscono delle parole allora devono essere inclusi prima della riga che contiene la direttiva `#include "../../../../generic/words.S"`.
- *user_main.c*: questo file è stato utile per aggiungere i propri file *header* C.

Per uno sviluppatore, è possibile aggiungere delle librerie esterne scritte in linguaggio C includendo innanzitutto i file header all'interno di *user_main.c*; se si volesse rendere disponibili queste funzionalità all'interprete di Punyforth, si devono esporre utilizzando la macro `defprimitive`. Questa macro (definita, come si è visto, all'interno del file *macros.S*) si occupa di definire direttamente in linguaggio *Assembly* delle parole, memorizzandole all'interno del dizionario, rendendole disponibili all'interprete esterno. Ad esempio, uno sviluppatore può esporre la funzione `int funzione_scritta_in_linguaggio_c(int a, int b)` tramite la parola `funzione` scrivendo il codice *Assembly* visibile nella sezione 9.1.6.

La `defprimitive` vuole specificati in questo ordine `name`, `namelen`, `label`, `flags`:

- `name` indica il nome della parola all'interno del dizionario;
- `namelen` indica la lunghezza del nome della parola che si sta definendo;
- `label` indica il nome disponibile in linguaggio *Assembly*;
- `flags` indica il tipo di parola (`REGULAR`, `HIDDEN`, `IMMEDIATE`).

La macro `DPOP <reg>` consuma un elemento dal *Data Stack* e lo inserisce all'interno del registro `<reg>`. La macro `CCALL <label>` effettua il salvataggio dello stato di alcuni registri all'interno dello *stack* puntato dal registro `sp`, poi effettua la chiamata alla funzione C specificata all'interno di `<label>` tramite la `call0`, ripristinando al termine lo stato dei registri salvati. Non si è trovata documentazione in merito al funzionamento della `call0`: a quanto si è potuto capire dai sorgenti e dalle prove effettuate, questa istruzione utilizza come parametri della funzione da chiamare i valori presenti all'interno dei registri che vanno da *a2* ad *a7*, ed inserisce i valori di ritorno all'interno degli stessi registri (si hanno più valori di ritorno quando viene restituita una *struct*). È possibile inserire in cima al *Data Stack* i valori ritornarti tramite la macro `DPUSH <reg>` che inserisci in cima al *Data Stack* il valore contenuto all'interno del registro `<reg>`.

Chiaramente, se si effettuano modifiche a questo livello, sarà necessario effettuare la compilazione dell'immagine binaria (i cui passi saranno mostrati nella sezione 5.5). Avendo a disposizione un interprete in grado di interpretare ed eseguire del codice simbolico, sicuramente quello esposto non è l'approccio migliore poiché richiede la ricompilazione dell'immagine binaria. È anche vero che molte librerie non sono disponibili in linguaggio FORTH (ma in linguaggio C, per cui andrebbero

riscritte in linguaggio FORTH); in questi casi può essere più conveniente (soprattutto per questioni di tempo legate alla traduzione da un linguaggio all'altro) l'approccio del *wrapper* Assembly illustrato in precedenza.

5.5 Compilazione

La compilazione di Punyforth, che produce l'immagine binaria *punyforth.bin*, richiede innanzitutto alcuni passi per la preparazione dell'ambiente di compilazione. Per la compilazione sono necessari due strumenti: *esp-open-sdk* (da cui si ottiene la *toolchain* di compilazione contenente il compilatore *xtensa-lx106-elf*) e *esp-open-rtos* (che contiene il sistema operativo su cui viene eseguito Punyforth).

Si premette che i passi illustrati fanno riferimento ad una distribuzione Linux basata su Debian. Detto ciò, il processo per la preparazione dell'ambiente di compilazione sono i seguenti:

1. Installare gli strumenti di compilazione e le dipendenze necessarie tramite il comando `apt-get`:

   ```
   $ sudo apt-get install make unrar-free autoconf automake
   libtool gcc g++ gperf flex bison texinfo gawk ncurses-dev
   libexpat-dev python-dev python python-serial sed git unzip
   bash help2man wget bzip2 libtool-bin
   ```

2. Creare una cartella (ad esempio *esp*, all'interno della cartella *home* dell'utente) in cui saranno scaricati gli strumenti:

   ```
   $ mkdir ~/esp
   $ cd ~/esp
   ```

3. Effettuare il *clone* di *esp-open-sdk* dal *repository GitHub https://github.com/pfalcon/esp-open-sdk.git*:

   ```
   $ git clone https://github.com/pfalcon/esp-open-sdk.git
   $ cd esp-open-sdk
   ```

4. Effettuare la compilazione, tramite `make`, degli strumenti che servono per la compilazione di *ESP Open RTOS* e di Punyforth:

   ```
   $ make toolchain esptool libhal STANDALONE=n
   ```

5. Aggiungere la cartella *~/esp/esp-open-sdk/xtensa-lx106-elf/bin* alla variabile d'ambiente `PATH`:

   ```
   $ export PATH=$PATH:~/esp/esp-open-sdk/xtensa-lx106-elf/bin
   ```

6. Effettuare il *clone* di *esp-open-rtos* dal *repository GitHub https://github.com/Superhouse/esp-open-rtos.git*:

   ```
   $ cd ..
   $ git clone --recursive https://github.com/Superhouse/esp-open-rtos.git
   ```

A questo punto, la cartella *~/esp* conterrà le due cartelle *esp-open-rtos* e *esp-open-sdk* e l'ambiente di compilazione sarà pronto occupando circa 4 GB all'interno del disco. Per compilare l'immagine binaria con all'interno Punyforth, bisogna:

1. Copiare la cartella contenente il progetto di Punyforth all'interno della cartella *~/esp/esp-open-rtos/examples*.
2. Entrare all'interno della cartella *~/esp/esp-open-rtos/examples/punyforth/arch/esp8266/rtos/user* che contiene i sorgenti e il file *Makefile* e lanciare il comando `make`.

Al termine del processo di compilazione, il file *punyforth.bin* (contenente l'immagine binaria) si troverà all'interno della cartella *~/esp/esp-open-rtos/examples/punyforth/arch/esp8266/rtos/user/firmware*. È sufficiente copiare questo file all'interno della cartella *~/esp/esp-open-rtos/examples/punyforth/arch/esp8266/bin* per usare lo strumento *flash.py* (al suo interno) per il download all'interno della flash.

Durante l'attività progettuale, si è preferito utilizzare un link simbolico alla cartella contenente Punyforth invece di copiarla direttamente all'interno della cartella *~/esp/esp-open-rtos/examples*.

6 Implementazione

All'interno dell'attività progettuale si sono seguiti approcci differenti per sviluppare strumenti con scopi differenti. Ciò che all'atto pratico si è implementato è stato, da una parte una serie di funzionalità per consentire l'integrazione e la comunicazione con il sistema di regole utilizzando stringhe di codice simbolico e codice scritto in linguaggio C; dall'altra parte, usando esclusivamente stringhe di codice simbolico, l'implementazione di strumenti utili alla configurazione e alla programmazione dei nodi della rete e l'implementazione di un client capace di utilizzare il protocollo *MQTT*, molto diffuso nell'ambito dell'IoT.

6.1 Modifiche al server TCP REPL di Punyforth

All'interno dell'attività progettuale il server TCP *REPL* (visto nella sezione 4.7.6) è stato modificato per venire incontro alle esigenze di programmazione dei nodi e di memorizzazione della configurazione, migliorando anche la sua stabilità e la velocità ottenibile dal suo utilizzo.

Come detto all'interno della sezione dedicata, la versione di base di questo modulo esegue l'interpretazione di ogni linea di codice simbolico non appena questa viene ricevuta dal *socket* TCP. Si ricorda che il *worker* all'interno del server TCP *REPL* utilizza la parola `netcon-readln` che legge dal *socket* i caratteri ricevuti, terminando l'operazione nel momento in cui incontra i caratteri `\r\n`; a questo punto il *buffer*, contenente i caratteri ricevuti, viene passato alla parola `eval` che interpreta la stringa di codice simbolico.

Durante l'utilizzo di questo modulo si è riscontrato un problema di sincronizzazione tra il *task* che si occupa della ricezione dei byte dal *socket* e il *task* che esegue l'interpretazione della stringa quando, sulla stessa connessione, si inviano più linee di codice simbolico alla massima velocità possibile di trasmissione; questa mancata sincronizzazione causa un accesso concorrente non controllato, da parte del processo produttore e di quello consumatore, al contenuto del *buffer* `line` di 128 byte utilizzato per la memorizzazione e per l'interpretazione della linea di codice simbolico, rendendo instabile ed incerto il risultato dell'interpretazione.

Per risolvere il problema, in prima battuta si era pensato di introdurre un ritardo sul client che invia le linee di codice simbolico (ad esempio tramite una funzione *sleep*). Il minimo ritardo trovato per cui si era ottenuto un risultato stabile è stato

quello di 220 millisecondi. Questo ritardo introdotto rallentava però l'intero processo di trasmissione, rendendo così il software il limite alla velocità di trasmissione e non le caratteristiche della rete; non era una soluzione ammissibile.

Si è pensato allora di introdurre un *buffer* (chiamato `buffer`) di 4096 KB. Lo scopo di questo *buffer* e quello di memorizzare le linee di codice simbolico ricevute dal *socket* TCP senza farle eseguire immediatamente dall'interprete. Nel momento in cui il client invia una linea vuota (vengono trasmessi i soli caratteri `\r\n`) il codice simbolico contenuto in `buffer` viene interpretato per intero. In questo modo, è stato possibile raggiungere la massima velocità con la totale stabilità del risultato.

L'introduzione di questo *buffer* all'interno del server TCP *REPL* ha reso possibile la memorizzazione del codice simbolico prima dell'effettiva esecuzione. Per fare ciò, la parola `command-loop` del server TCP *REPL* di Punyforth è stata modificata. In base al contenuto di `line` il comportamento ottenuto è diverso:

- Se `line` contiene esattamente la parola *quit*: come nella versione originale di questo modulo, si procede all'uscita dalla parola `command-loop` per chiudere il *socket* con il client connesso.
- Se `line` contiene esattamente la parola *undo*: il contenuto di `buffer` viene svuotato. Consente di revocare ciò che si era precedentemente inviato. Per eseguire quest'operazione, viene usata la parola `clr`.
- Se `line` contiene esattamente la parola *store*: il codice simbolico contenuto in `buffer` non viene eseguito, ma viene memorizzato all'intero del blocco di flash SPI che si è riservato per la memorizzazione del codice simbolico dell'utente. Per eseguire quest'operazione, viene usata la parola `conf!`.
- Se `line` è vuota: come detto in precedenza, si esegue l'interpretazione del codice simbolico contenuto in `buffer`. `buffer` viene svuotato.
- Se non si è verificato uno dei casi precedenti: i caratteri ricevuti rappresentano del codice simbolico che viene man mano memorizzato accodandolo all'interno di `buffer`. Per eseguire quest'operazione, viene usata la parola `buf!`.

Il funzionamento della parola `command-loop` è illustrato in Figura 13. Il client invia sul *socket* alcune stringhe di codice simbolico (`SYMBCODE1, SYMBCODE2, ..., SYMBCODEn`) che vengono memorizzate in `buffer` per poi essere trattate in base all'operazione richiesta successivamente.

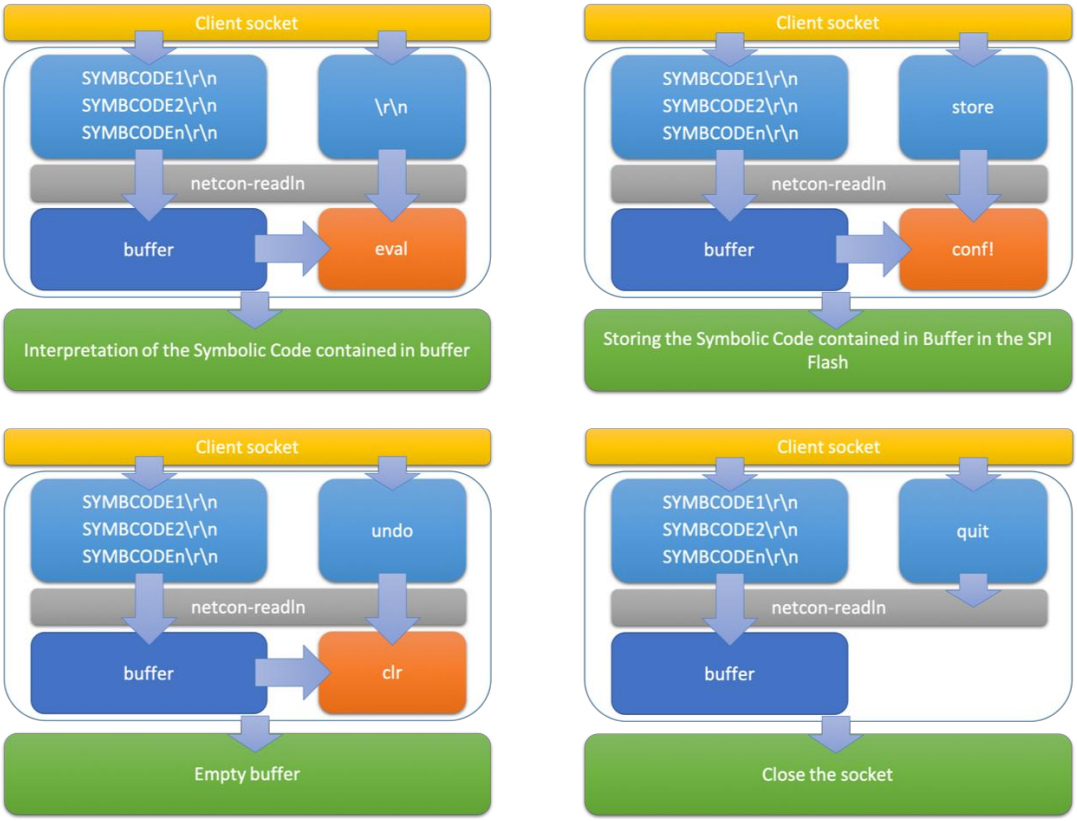

Figura 13 - TCP REPL - Funzionamento della parola command-loop

6.2 Interazione con il sistema di regole

Il protocollo applicativo implementato per l'interazione tra il sistema di regole (scritto in Prolog) e i nodi della rete (su cui è presente l'interprete FORTH) si basa sull'invio di stringhe di codice simbolico tra le varie componenti.

Il server TCP *REPL*, presente all'interno dell'MCU, fornisce all'utente la possibilità di usare una *shell* remota su Punyforth. L'idea che sta alla base di ciò che è stato implementato in questa sezione, è quella secondo cui è il sistema di regole stesso ad aprire una connessione verso i vari nodi della rete, mettendo a disposizione dell'utente un client per trasmettere stringhe di codice simbolico per la configurazione dei nodi o l'esecuzione su di essi di operazioni per il raggiungimento dell'obiettivo prefissato secondo una determinata pianificazione effettuata dall'utente stesso.

Il sistema di regole utilizza una base di conoscenza costituita da *fatti* e *regole*, sotto forma di *clausole di Horn*, per la definizione delle caratteristiche dei nodi sia a livello di *BOARD* (quindi i PIN con le loro caratteristiche, effettuando una descrizione dei sensori e attuatori ad essi connessi) che a livello di *NETWORK*

(rappresentando quindi le informazioni utili per la comunicazione da remoto con i vari nodi - ad esempio nome *host* o indirizzo IP, numero della porta del server *REPL*).

La parte dedicata alla *BOARD*, quindi, mette il sistema di regole a conoscenza di tutti gli spazi di configurazione possibili e quindi delle funzioni che possono essere eseguite dai vari nodi; ad esempio, se un nodo ha collegato ai PIN della sua *board* un sensore in grado di misurare la luminosità di una stanza e un *relay* utilizzato per controllare una lampada, su di esso si possono eseguire le operazioni dedicate al monitoraggio e al controllo della luminosità tramite l'utilizzo di codice simbolico; il codice, opportunamente prodotto, è inviato al nodo dal sistema di regole; il codice simbolico, interpretato in tempo reale dal nodo, restituisce il valore misurato dal sistema di regole che si occupa di confrontarlo con un determinato valore di soglia per poi, in base al risultato ottenuto dal confronto, eseguire l'operazione opportuna inviando al nodo il codice simbolico utile ad accendere o spegnere la lampada attraverso il relay.

Per consentire la comunicazione con i nodi, la base di conoscenza contiene al suo interno le informazioni per raggiungerli tramite la rete basata su IP; si definisce un fatto tramite `mcu_net_address/3`:

```
mcu_net_address(myesp, 'myesp.host.name', 1983)
```

In questo modo si sta dicendo al sistema di regole che il nodo chiamato `myesp` è raggiungibile all'interno della rete all'indirizzo ottenibile dalla risoluzione del nome di dominio `myesp.host.name` e che il server TCP *REPL* del nodo è in ascolto sulla porta `1983`. Tramite la regola `mcu_send_message/3` è possibile inviare un messaggio contenente del codice simbolico al nodo specificato e ricevere una risposta:

```
mcu_send_message(myesp, MessagesList, ResponseList)
```

In questo modo, le stringhe presenti all'interno della lista `MessagesList` vengono inviate al nodo `myesp` e le risposte ricevute vengono accodate alla lista `ResponseList`. Questa regola individua l'indirizzo IP del nodo `myesp` e la porta del server TCP *REPL* tramite il fatto `mcu_net_address` sopra descritto; le variabili popolate dalla risoluzione di questo fatto sono utilizzate dalla regola `net_communication/4` che si occupa di aprire un *socket* TCP e di lanciare un *thread* (per non bloccare quello principale di Prolog) che si occupa di inviare, uno alla volta, i messaggi contenuti all'interno della lista `MessagesList` accodando le risposte ricevute all'interno della lista `ResponseList`. Alla ricezione dell'ultimo

messaggio di risposta, la regola chiude il *socket* TCP aperto terminando la connessione e rendendo disponibili, per le regole successive, i messaggi contenenti i dati per essere mostrati o per effettuare una loro valutazione per potere eseguire le scelte che determineranno le operazioni successive. Esiste anche un'altra versione della regola `mcu_send_message`, con *arità* 2, che non prevede la ricezione dei messaggi dal server TCP *REPL*: in questo caso il *thread* avviato per la comunicazione non rimane in attesa di risposte dal server (contenenti messaggi di codice simbolico), ma termina non appena completa con successo l'invio dei messaggi verso il nodo.

Ad esempio, supponiamo che l'utente, tramite il sistema di regole, volesse mandare al nodo `myesp` la stringa di codice simbolico `LAMP ON` per accendere una lampada (collegata al *relay* che a sua volta è collegato ad uno dei PIN della ESP8266-12E, opportunamente memorizzato all'interno della costante `LAMP` all'interno del dizionario FORTH) e la stringa `LUM GET` per ottenere un feedback sull'operazione richiesta tramite un sensore di luminosità analogico (collegato al PIN *ADC* identificato dalla parola `LUM`); l'utente può eseguire questa azione usando la

```
mcu_send_message(myesp, ["LAMP ON LUM GET .\r\n"], Risp)
```

sul sistema di regole per inviare al nodo la stringa di codice simbolico che, se valida, sarà eseguita portando a termine le operazioni richieste. Infatti, sul nodo, il server TCP *REPL* riceve la stringa inviata dal client; questa stringa viene interpretata tramite la parola `eval` che esegue l'accensione della lampada `LAMP` usando la parola `ON` e la misurazione della luminosità tramite la parola `GET` utilizzando la parola `adc-read` per leggere il valore di luminosità (dal sensore analogico); il valore ottenuto viene inserito in cima la *Data Stack*; infine, il valore misurato viene inviato al client tramite la parola `.` scrivendo sul *socket* (aperto per il client Prolog) il valore di luminosità consumandolo dal *Data Stack*. L'intero processo è schematizzato in Figura 14.

Il protocollo applicativo implementato all'interno base di conoscenza è quello su cui si basa il server TCP *REPL* di Punyforth con le modifiche viste nella sezione 6.1.

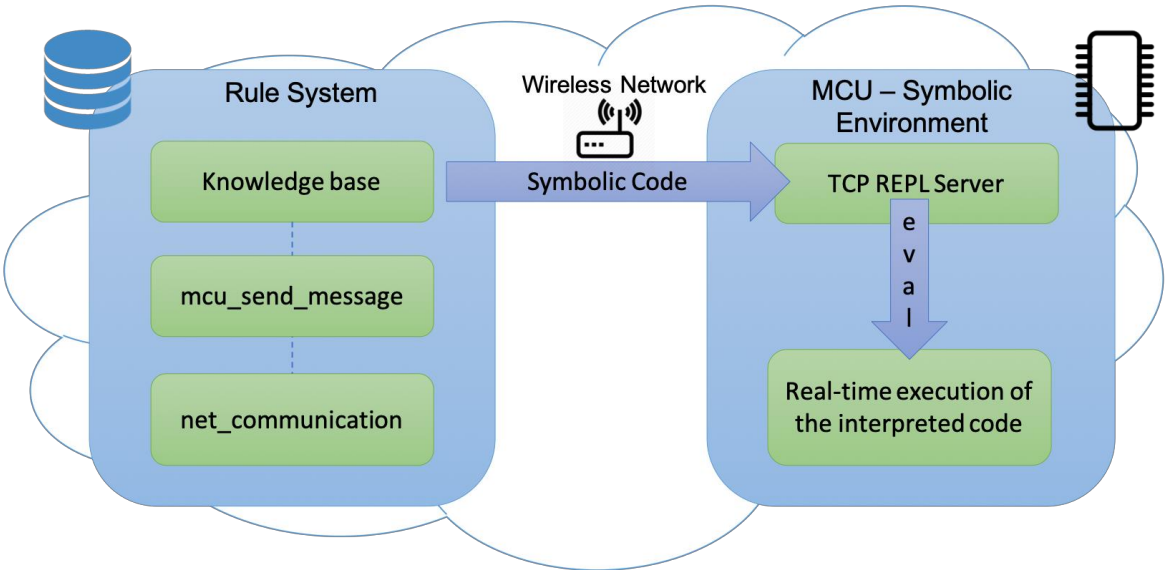

Figura 14 - Interazione Sistema di Regole e Ambiente Simbolico

6.3 Interazione sicura con il sistema di regole

I dispositivi che possono essere gestiti da ciò che si è implementato possono essere di vario tipo e possono consentire la raccolta di dati sensibili e l'esecuzione di azioni che possono rilevarsi potenzialmente pericolose in base al contesto d'uso. Basti pensare, ad esempio, ad un sistema per il monitoraggio e il controllo dei dati ambientali di un *Centro Elaborazione Dati* (*CED*) di una media/grande azienda in cui sono presenti diversi sensori ed attuatori utili per il mantenimento dei parametri di umidità e di temperatura al di sotto di una soglia critica: se qualcuno fosse in grado di potere comunicare liberamente con i nodi di questa rete, potrebbe facilmente alterare le condizioni ambientali del *CED* compromettendo l'efficienza delle macchine presenti al suo interno e la perdita di dati importanti.

L'esempio descritto sopra, mette in evidenza la necessità di avere a disposizione un canale sicuro per la trasmissione delle stringhe di codice simbolico tra i nodi. È vero che gli MCU come la ESP8266-12E sono collegate alla rete tramite Wi-Fi utilizzando i protocolli standard di cifratura, ma la cifratura sarà effettuata solo all'interno della *BSS*: una volta che il messaggio raggiungerà l'*Access Point* per l'instradamento sulla rete cablata (salvo la presenza di altri strumenti per la cifratura a livello di rete come *VPN* o *IPsec*), questo sarà immesso in chiaro all'interno della rete verso il nodo di destinazione. Un potenziale attaccante, collegato alla rete, potrebbe quindi ottenere informazioni ed eseguire azioni senza alcun controllo.

Per gestire questa minaccia, si è pensato di potenziare il protocollo applicativo di comunicazione (descritto nella sezione 6.1) tramite l'implementazione di un algoritmo di cifratura in modo che i vari nodi della rete possano scambiarsi messaggi criptati con delle chiavi simmetriche conosciute solo dai vari interlocutori della rete. Per raggiungere questo scopo, si sono implementati gli algoritmi di cifratura *DES-CBC* e *AES128-CBC*. La scelta di DES-CBC ricade sulla minore complessità computazionale rispetto ad AES128-CBC; la scelta di AES128-CBC ricade sulla sua maggiore robustezza rispetto a DES-CBC.

L'applicazione di questi algoritmi di cifratura ha messo in evidenza un problema da gestire: il protocollo applicativo descritto sopra utilizza come *token* i simboli *CR LF* per identificare la fine di una linea e il simbolo *NUL* per identificare la fine di una stringa; detto ciò, lo spazio dei simboli utilizzati dagli algoritmi di cifratura scelti per l'attività progettuale si muove nell'intervallo 0 - 255, intervallo in cui sono codificati anche i caratteri non stampabili sopra descritti e fondamentali per il funzionamento del protocollo applicativo. Se il messaggio criptato ottenuto contenesse questi simboli nel posto sbagliato, le funzioni dedicate alla lettura dei caratteri dal *socket* spezzerebbero il messaggio in modo casuale e non previsto. Per gestire questo problema si è pensato di utilizzare l'algoritmo *Base64* per la codifica dei messaggi criptati prima di essere inviati. Il processo è mostrato in Figura 15.

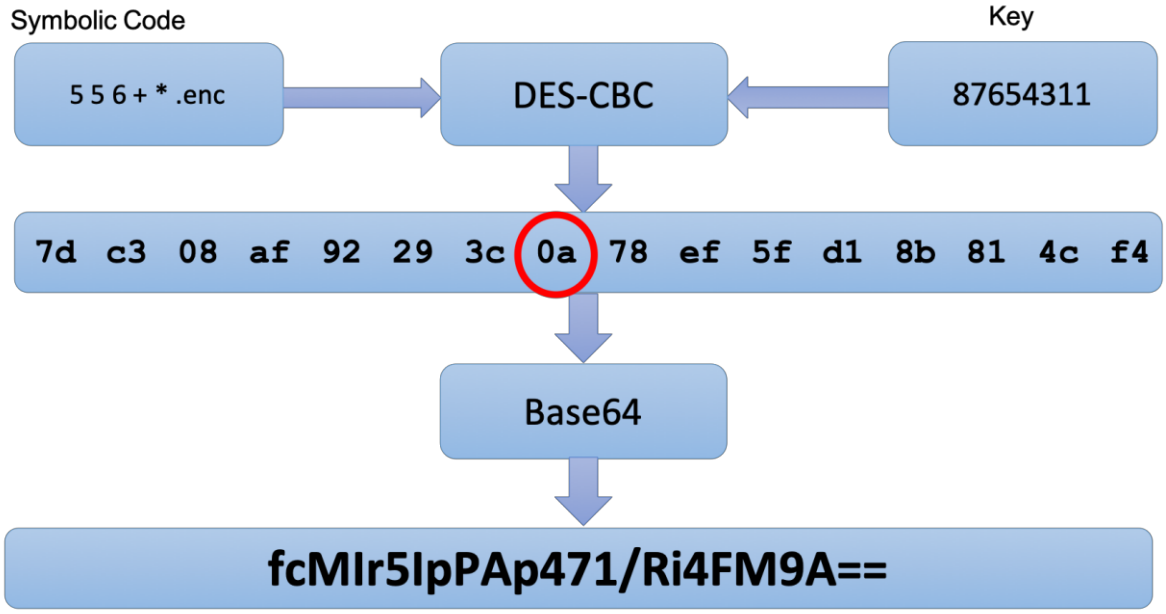

Figura 15 - Cifratura e Codifica di una stringa di codice simbolico utilizzando DES-CBC e Base64

Per quanto riguarda l'implementazione, si è scelto di inserire all'interno della memoria flash SPI dei nodi della rete una *Master Key* (da rendere nota solo al diretto

possessore/amministratore del sistema), da inserire manualmente all'interno della base di conoscenza del sistema di regole, e da utilizzare in fase di legame tra i dispositivi; il protocollo prevede l'uso di una trasmissione basata su chiavi *master* simmetriche (unica per ogni nodo) per lo scambio di chiavi di sessione, anch'esse simmetriche, utilizzate nelle effettive comunicazioni. L'uso di protocolli a chiavi asimmetriche (come l'*RSA*) per lo scambio di chiavi simmetriche di sessione, sarebbe stata una scelta sicuramente migliore in termini di sicurezza; la ESP8266-12E utilizzata durante l'attività progettuale è un dispositivo con risorse limitate: a causa di limiti in termini di spazio in memoria che di prestazioni non è stato possibile procedere con la sua implementazione.

Ogni comunicazione da parte del sistema di regole con i vari nodi è autenticata dal fatto che ogni chiave è specifica per nodo; il sistema di regole ha la certezza di comunicare con uno specifico nodo in quanto è considerato l'unico in grado di decifrare, interpretare e reagire correttamente al messaggio inviato. Infatti, la decifratura non corretta di un messaggio darebbe luogo ad una stringa di codice simbolico non valida che conterrà dei simboli che l'interprete non sarà in grado di interpretare.

6.3.1 Implementazione all'interno dell'ambiente simbolico

Come anticipato nell'introduzione di questo capitolo, si è provveduto all'implementazione utilizzando delle librerie scritte in linguaggio C contenenti il codice sorgente per il funzionamento degli algoritmi di cifratura DES-CBC e AES128-CBC e dell'algoritmo di codifica Base64. Le funzioni, utili ad avviare i vari processi di cifratura/decifratura e codifica/decodifica, sono state esposte verso l'interprete esterno di Punyforth tramite il *wrapper* scritto in codice *Assembly*, utilizzando il lessico e la sintassi che si sono visti nelle sezioni precedenti; in questo modo, tramite degli appositi simboli definiti, è possibile eseguire sul nodo le varie operazioni direttamente dalla *shell* locale o tramite l'accesso remoto al nodo.

Dopo avere testato ed effettuato il *debug* su architetture x86 i sorgenti scritti in linguaggio C delle librerie utilizzate, queste sono state copiate all'interno della cartella *arch/esp8266/rtos/user* di Punyforth. Quindi, oltre ai sorgenti già presenti nel progetto di Punyforth, si trovano anche i file *aes.c*, *aes.h*, *des.c*, *des.h*, *base64.c*, *base64.h* contenenti le implementazioni in linguaggio C degli algoritmi interessati (i sorgenti sono visibili all'interno dell'Appendice alla sezione 9.2).

Successivamente, si è proceduto a scrivere il *wrapper* in Assembly contenente le definizioni per utilizzare gli algoritmi utilizzati direttamente come codice simbolico. Si sono definite le seguenti parole nei file *aes.S* e *des.S*:

- `>aes ( ciphertext_buffer plaintext_buffer key plaintext_length -- ciphertext_length ):` questa parola ha il compito di richiamare la funzione C per effettuare la cifratura di una stringa utilizzando l'algoritmo AES128-CBC

  ```
  uint32_t AES_CBC_encrypt_buffer(char *output, char *input, char *key, int length)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*, *a4*, *a5*. La funzione restituisce un intero rappresentante la lunghezza della stringa criptata che viene inserito in cima al *Data Stack* al termine dell'esecuzione della parola. Il risultato della cifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `ciphertext_buffer`.

- `aes> ( plaintext_buffer ciphertext_buffer key ciphertext_length -- plaintext_length )`: questa parola, ha il compito di richiamare la funzione C per effettuare la decifratura di una stringa utilizzando l'algoritmo AES128-CBC

  ```
  uint32_t AES_CBC_decrypt_buffer(char *output, char *input, char *key, int length)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*, *a4*, *a5*. La funzione restituisce un intero rappresentante la lunghezza della stringa decriptata che viene inserito in cima al *Data Stack* al termine dell'esecuzione della parola. Il risultato della decifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `plaintext_buffer`.

- `>des ( ciphertext_buffer plaintext_buffer key plaintext_length -- ciphertext_length ):` questa parola ha il compito di richiamare la funzione C per effettuare la cifratura di una stringa utilizzando l'algoritmo DES-CBC

  ```
  uint32_t des_encrypt(uint8_t *output, uint8_t *input, uint8_t* key, uint16_t length)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*, *a4*, *a5*. La funzione restituisce un intero rappresentante la lunghezza della stringa criptata

che viene inserita in cima al *Data Stack* al termine dell'esecuzione della parola. Il risultato della cifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `ciphertext_buffer`.

- `des> ( plaintext_buffer ciphertext_buffer key ciphertext_length -- plaintext_length )`: questa parola, ha il compito di richiamare la funzione C per effettuare la decifratura di una stringa utilizzando l'algoritmo DES-CBC

  ```
  uint32_t des_decrypt(uint8_t *output, uint8_t *input, uint8_t* key, uint16_t length)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*, *a4*, *a5*. La funzione restituisce un intero rappresentante la lunghezza della stringa decriptata che viene inserito in cima al *Data Stack* al termine dell'esecuzione della parola. Il risultato della decifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `plaintext_buffer`.

- `>base64 ( digest_buffer plaintext_buffer plaintext_length -- )`: questa parola ha il compito di richiamare la funzione C per effettuare la codifica di una stringa in Base64

  ```
  Base64encode(char * coded_dst, const char *plain_src, int len)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*, *a4*. Il risultato della cifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `digest_buffer`.

- `base64> ( plaintext_buffer digest_buffer -- ):` questa parola ha il compito di richiamare la funzione C per effettuare la decodifica di una stringa codificata in Base64

  ```
  void Base64decode(char * plain_dst, const char *coded_src)
  ```

  i cui parametri saranno presenti all'interno dei registri *a2*, *a3*. Il risultato della cifratura è presente all'interno del *buffer* puntato dall'indirizzo di memoria contenuto in `plaintext_buffer`.

Fatto ciò, i file *header* del C sono stati inclusi all'interno del file *user_main.c* tramite la direttiva `#include` e i file *Assembly* sono stati inclusi all'interno del file *punyforth.S* tramite la direttiva `#include` prima dell'*include* del file *words.S*. Per la

poca disponibilità di memoria del dispositivo utilizzato, non è stato possibile includere entrambi gli algoritmi di cifratura contemporaneamente. Come si è potuto vedere, sia le parole per l'AES128-CBC che quelle per il DES-CBC sono state definite per consumare e rilasciare esattamente gli stessi parametri e nello stesso ordine dal *Data Stack* in modo da essere usate alla stesse maniera. Terminata questa fase, si è effettuata la compilazione dell'immagine binaria per aggiungere le nuove funzionalità implementate; poi si sono effettuate le modifiche al codice FORTH del server TCP *REPL*.

All'interno del file *tcp-repl.forth* sono state fatte delle modifiche alla parola `command-loop` per potere gestire il nuovo protocollo di comunicazione con l'algoritmo di cifratura/decifratura e l'algoritmo di codifica/decodifica. La modalità precedente del protocollo è stata comunque lasciata e, tramite il valore contenuto all'interno della variabile `cripto-mode`, è possibile scegliere se utilizzare o meno il protocollo di comunicazione di base o quello con gli algoritmi per la sicurezza, rispettivamente se il suo valore sarà `0` o `-1`. In questa versione del protocollo di comunicazione, all'avvio di Punyforth questa variabile è settata al valore `-1`. L'utente può cambiare la modalità in *run-time* tramite le parole `+des` e `-des` che, rispettivamente, abilitano e disabilitano la modalità che prevede l'uso dell'algoritmo di sicurezza fino al prossimo riavvio del nodo. La chiave simmetrica viene memorizzata all'interno del *buffer* `key` e può essere modificata *run-time* con le varie chiavi di sessione tramite la parola `setkey`.

Il codice FORTH aggiunto a `command-loop` si occupa di ricevere la stringa criptata, suddivisa in blocchi (la cui dimensione è definita all'interno della costante `BLOCKLEN`) e codificata in Base64 dal client; ogni linea inviata dal client, corrisponde ad un blocco esatto codificato interamente in Base64; non appena una linea viene ricevuta, la procedura esegue

```
b64cline line base64>
```

che decodifica il contenuto di `line`, codificato in `Base64`, e lo memorizza all'interno del buffer `b64cline` che, a questo punto, contiene esattamente `BLOCKLEN` caratteri criptati secondo l'algoritmo di cifratura in uso; immediatamente dopo, tramite

```
BLOCKLEN counter @ * encline + b64cline BLOCKLEN copyc
counter @ 1 + counter !
```

questi caratteri vengono accodati all'interno del *buffer* `encline` che, alla fine della procedura, contiene l'intero messaggio criptato. La procedura continua ad accodare i simboli finché la linea ricevuta non conterrà, esattamente nelle posizioni 0 e 1, il carattere `0x04` (`EOT`). A questo punto, se il *buffer* `encline` contiene dei caratteri, si esegue

```
decline encline key BLOCKLEN counter @ * des> drop
```

che effettua la decifratura (tramite DES-CBC nel caso d'esempio) dei caratteri contenuti all'interno del *buffer* `encline` memorizzando il risultato all'interno del *buffer* `decline` e scartando (tramite `drop`) la dimensione della risposta decriptata. Se la dimensione della stringa contenuta all'interno di `decline` è maggiore di zero, questa viene interpretata dalla parola `eval` eseguendo l'operazione desiderata.

Si esprimono delle considerazioni a riguardo. In prima battuta si era pensato di implementare questa parte utilizzando esclusivamente codice simbolico; questo primo approccio ha rilevato un grosso limite di Punyforth: come detto nella sezione 5.4, tutto ciò che viene definito dall'interprete di Punyforth viene collocato all'interno dell'*heap* riservato per Punyforth che, di predefinito, si attesta sui 25 Kb. La grande dimensione delle tabelle utilizzate dagli algoritmi AES128-CBC, DES-CBC e Base64, una volta passate dall'interprete di Punyforth, vanno a saturare lo spazio disponibile per le nuove definizioni, rendendo instabile il dispositivo e impossibile da utilizzare. Aggiungendo queste librerie direttamente all'interno dell'immagine binaria, si è lasciato il compito di allocazione in memoria direttamente al sistema operativo che non è andato ad occupare l'*heap* riservato a Punyforth, consentendone il normale utilizzo. Si era pensato anche ad un altro approccio: poiché la ESP8266-12E supporta il Wi-Fi, al suo interno dovrebbe avere, di conseguenza, un'implementazione dell'AES e del DES. Questo è vero, ma non si è riusciti, nonostante la versione "*open*" dell'SDK e dell'*RTOS*, ad individuare gli algoritmi utilizzabili al di fuori del contesto di cifratura sulle reti Wi-Fi.

Detto ciò, si è comunque pensato di escludere queste funzionalità per un motivo di performance: dalle prove sperimentali eseguite si è notato che l'utilizzo di DES-CBC ha rallentato il protocollo di comunicazione di 2 volte rispetto a quello predefinito; l'utilizzo di AES128-CBC ha rallentato il protocollo di comunicazione addirittura di 4 volte rispetto a quello predefinito.

6.4 Comunicazione tra nodi tramite TCP REPL e TCP Client

Nell'implementazione precedente dell'attività progettuale svolta, basata prettamente sul sistema di regole, i vari nodi della rete non potevano comunicare direttamente tra loro ed ogni loro azione era coordinata dal sistema di regole centralizzato. Si è pensato allora di rendere i nodi indipendenti da un sistema centralizzato e capaci di inviare e ricevere stringhe di codice simbolico in maniera indipendente, implementando un client TCP direttamente in codice simbolico. In questo modo, ogni nodo mantiene in esecuzione un server TCP *REPL* in ascolto delle richieste dei vari nodi e un client TCP in grado di inviare richieste agli altri nodi.

Per fare ciò, si è implementato un nuovo modulo chiamato *tcp-client.forth* (consultabile alla sezione 9.2.10). All'interno di questo modulo si è definita la variabile `socket` per memorizzare il *socket* `netcon` (utilizzato per la comunicazione con il nodo server) e un *task* contenuto all'interno di `netfetch-task`. Il modulo è molto semplice e contiene le definizioni delle parole utili per effettuare la connessione all'altro MCU, l'invio di codice simbolico (con ricezione dell'eventuale risposta) e la chiusura della connessione:

- `tcp-connect ( port ip -- )`: questa parola apre la connessione verso il nodo server all'indirizzo contenuto dentro `ip` e alla porta `port`. Se il *socket* viene aperto, viene memorizzato all'interno della variabile `socket`. A questo punto, usando la parola `tcp-receive`, parte il *task* che rimane in ascolto dei messaggi di risposta, lasciando libero il *task* principale per eseguire altro codice simbolico.
- `tcp-receive ( -- )`: questa parola è utilizzata nel *task* `netfetch-task` ed è quella che si occupa di leggere i caratteri (ricevuti dal nodo server) dal *socket* `socket` tramite la parola `netcon-readln` memorizzandoli all'interno del *buffer* `line`; se il *socket* non è stato chiuso dal nodo server (la `netcon-readln`, in questo caso inserisce in cima al *Data Stack* l'interno `-1`) e se la stringa contenuta all'interno di `line` non è *quit*, la stringa `line` viene passata all'interprete tramite la parola `eval` eseguendo il codice simbolico inviato dal nodo server. Nel caso in cui, invece, la stringa contenuta in `line` è *quit* o il *socket* è stato chiuso dal server, il client TCP libera il *socket* chiudendo la connessione.

- `tcp-send ( str -- )`: questa parola manda il contenuto della stringa `str` al nodo server scrivendolo sul *socket* `netcon` tramite la parola `netcon-write`.
- `tcp-disconnect ( -- )`: questa parola usa la parola `tcp-send` per inviare la stringa *quit* al nodo server, informandolo così che il client desidera chiudere la connessione.
- `tcp-response ( -- )`: questa parola viene inviata dal client al server sotto forma di codice simbolico. Il server, eseguendo questa parola, trasmette al client il seguente codice simbolico:

  `tcpcli-mailbox mailbox-send`

  In questo modo, un altro *task* in attesa della mailbox `tcpcli-mailbox` verrebbe sbloccato per eseguire il codice simbolico dedicato a gestire la risposta del server.
- `tcp`: questa parola memorizza l'inizio dell'array utilizzato per la memorizzazione degli *execution token* di `tcp-connect tcp-disconnect tcp-send`. Combinata con le parole `connect`, `disconnect` e `send` (definite in *utils.forth*) consente l'esecuzione delle varie funzionalità precedentemente descritte.

Lo scambio di codice simbolico tra due MCU, utilizzato il client TCP e il server TCP *REPL*, è mostrato in Figura 16.

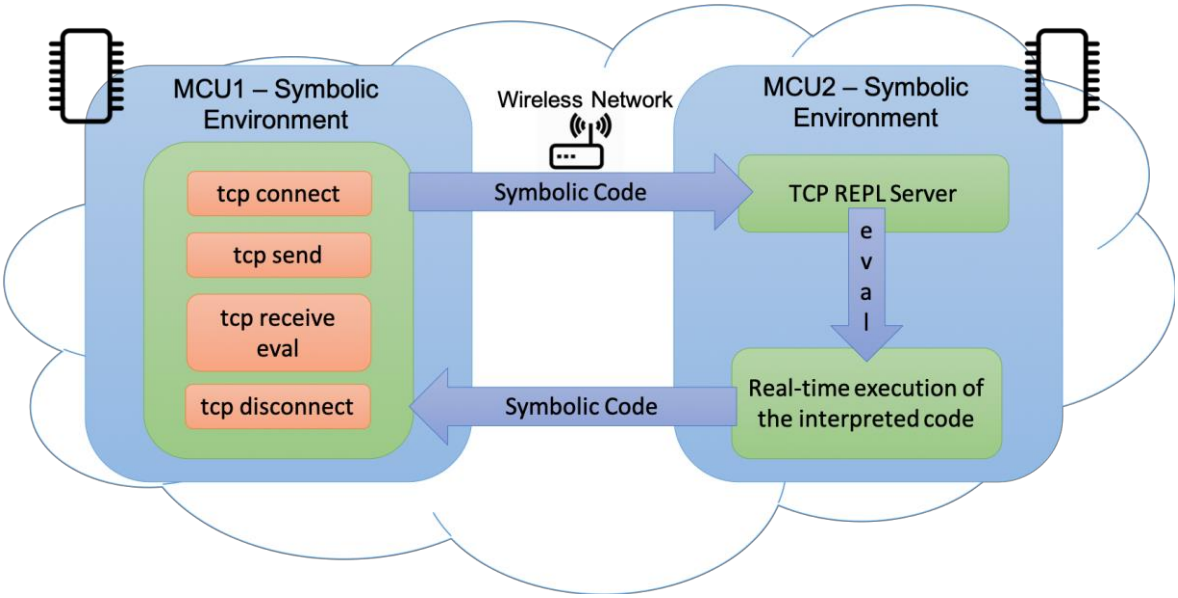

Figura 16 - Interazione Client TCP e Server TCP REPL

6.5 Comunicazione tra nodi tramite MQTT

L'*MQTT* (*Message Queue Telemetry Transport*) è un protocollo a livello di applicazione introdotto da *IBM* nel 1999 e reso standard da *OASIS* nel 2013 [7]. Il compito per cui è stato ideato è quello di consentire la connettività e la comunicazione attraverso la rete tra applicazioni e *middleware* consentendo l'interazione e lo scambio di informazioni tra piattaforme diverse. Questo protocollo è stato progettato per essere semplice, leggero e facile da implementare; per questi motivi, questo protocollo è adatto per ambienti in cui vi sono dispositivi a risorse limitate, dove i costi della rete sono molto elevati o dove la rete ha una banda limitata [19].

La sua architettura (vedi Figura 17) si basa sul modello del *publisher*/*subscriber* secondo cui si distinguono:

- *Publisher*: i mittenti dei messaggi.
- *Subscriber*: i destinatari dei messaggi.
- *Broker*: si occupa di consegnare ai *Subscriber* i messaggi inviati dai *Publisher*.

Un *Subscriber* si registra su un *Broker* indicando un *Topic* su cui vuole ricevere informazioni. Un *Topic* è una stringa che consente al *Broker* di filtrare i messaggi da inviare ad ogni *Subscriber*; la sua struttura è quella di un percorso a più livelli e ogni livello (chiamato livello di *Topic*) è separato dal simbolo `/`. Ad esempio

`casa/cucina/luminosita`

`casa/salone/luminosita`

`casa/cucina/tavolo/lampada/1`

`casa/cucina/tavolo/lampada/2`

sono *Topic* validi su cui un *Subscriber* può registrarsi. Un *Topic* deve contenere almeno un carattere e, al suo interno, può contenere anche spazi vuoti. *Publisher* e *Subscriber* possono riferirsi ai *Topic* in due modi differenti: specificando il *Topic* esatto oppure usando delle *Wildcard* che, a loro volta si distinguono in *Wildcard* a singolo livello, identificate dall'uso del simbolo `+`, e in *Wildcard* a livello multiplo, identificate dal simbolo `#`. Secondo gli esempi precedenti di *Topic*, se l'utente volesse conoscere lo stato di luminosità delle varie stanze della `casa`, può inviare un messaggio al *Topic* `casa/+/luminosita`; oppure, se volesse mandare un messaggio a tutti i dispositivi in `cucina`, può inviare un messaggio al *Topic* `casa/cucina/#`.

Questa breve descrizione dell'*MQTT* fa capire anche le sue potenzialità descrittive all'interno dell'IoT: è possibile effettuare una descrizione gerarchica dell'ambiente e riferirsi ai nodi della rete tramite il loro significato. Questo è uno dei motivi per cui si è scelto di implementare questo protocollo all'interno dell'attività progettuale risultando, teoricamente, anche di facile integrazione all'interno del sistema di regole attualmente implementato.

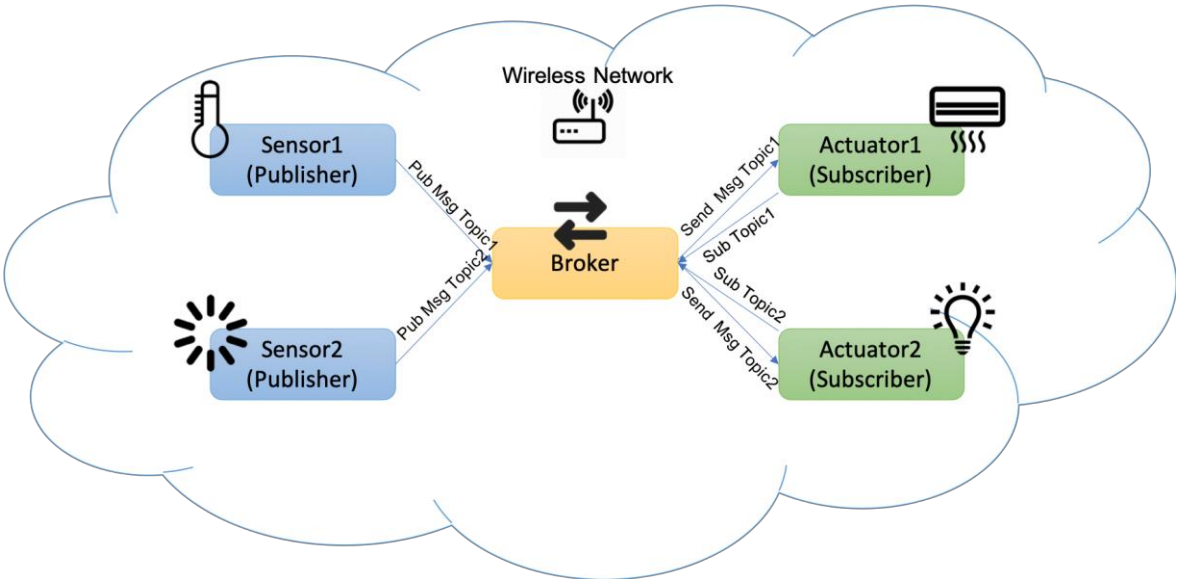

Figura 17 - Scambio di messaggi tramite il protocollo MQTT

6.5.1 Implementazione all'interno dell'ambiente simbolico

Vi sono diverse librerie, disponibili sul Web e scritte in linguaggio C per la versione di *RTOS* della ESP8266, che implementano un client *MQTT*. Queste librerie, come ad esempio la *ESP-RTOS-MQTT* (presente su *https://github.com/vaibhav93/ESP-RTOS-MQTT*) mettono a disposizione dello sviluppatore un client *MQTT* pronto all'uso da includere all'interno del progetto ricompilando l'immagine binaria; si è scelto comunque di implementare un client *MQTT* usando direttamente il codice simbolico del linguaggio FORTH per alcuni semplici motivi: queste librerie scritte in linguaggio C sono molto articolate (l'implementazione del client *MQTT* in linguaggio C è effettuata usando diverse migliaia di linee di codice) e la loro compilazione non è molto intuitiva. L'implementazione eseguita in codice FORTH durante l'attività progettuale ha consentito la sua scrittura in meno di 140 linee di codice senza la necessità di effettuare la ricompilazione del codice sorgente, dato che questo viene interpretato. È vero che questo modulo non rappresenta l'implementazione completa del client *MQTT*, ma fornisce un client *MQTT* funzionante (consultabile alla sezione 9.2.11).

Per implementare il client *MQTT*, si è preso come riferimento il manuale, scritto da *IBM*, con le specifiche della versione 3.1 del protocollo [20]. Il protocollo stabilisce un formato per il messaggio (costituito da un *header* e da un *body*) e una serie di comandi. L'*header* di ciascun comando è composto da un *header fisso* e, opzionalmente, da un *header variabile*. L'*header* fisso, come si può vedere dalla Tabella 8, è costituito da 2 byte. Il byte 1 contiene nei bit 7-4 il *Message Type* che identifica uno dei possibili comandi (elenco presente in Tabella 9); i 4 bit rimanenti rappresentano dei *flag*. Il byte 2 contiene la *Remaining Length* che rappresenta il numero di byte successivi che comprendono l'*header* variabile e il *body*, che rappresenta il reale *payload* del messaggio. Si rimanda al manuale del protocollo [20] per maggiori dettagli.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

Tabella 8 - Header fisso del protocollo MQTT

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

Tabella 9 - Comandi utilizzati dal MQTT

Il client *MQTT* è stato implementato usando le seguenti parole:

- `mqtt-connect ( port ip -- )`: effettua la connessione ad un *Broker* usando `ip` e `port`. Come si può vedere dal codice, si inizializza un `buffer` con i byte utili per eseguire la procedura di connessione.

  I primi due byte rappresentano l'*header* fisso: il byte 1 è impostato a `0x10` per il comando `CONNECT`; il byte 2 è impostato a `0x12` per indicare che la parte rimanente del messaggio è lunga 18 byte. Da qui parte l'*header* variabile: i due byte successivi, `0x00` e `0x06`, indicano la lunghezza della stringa che identifica il tipo di protocollo; i prossimi 6 byte contengono la stringa `MQIsdp` (indica che si sta utilizzando l'*MQTT*); il byte successivo, `0x03`, indica la versione del protocollo; il byte successivo, `0x02`, indica che si sta iniziando una nuova sessione; i due byte successivi, `0x00` e `0x3C`, impostano il timer di *keep alive* a 60 secondi.

  La parte rimanente è il *payload* del commando `CONNECT`: i primi due byte, `0x00` e `0x04`, indicano la sua lunghezza; i 4 byte rimanenti contengo la stringa `esp1` che identifica il client all'interno del *Broker*.

  Le parole successive si occupano di aprire il *socket* per la connessione, di inviare il messaggio contenuto all'interno di `buffer` e di consumare dal *socket* la risposta dal *Broker* che, alla ricezione della richiesta di connessione, esegue il comando `CONNACK` che prevede un messaggio lungo 4 byte (in questa versione la risposta non viene esaminata). Al termine il *socket* `netcon` viene memorizzato all'interno della variabile `mqtt-socket`; nel caso in cui ci fosse qualche problema di connessione, l'eventuale eccezione lanciata viene catturata e mostrando un messaggio di errore sull'*output buffer*.

- `mqtt-disconnect ( -- )`: questa parola prevede l'invio di un comando più semplice rispetto al precedente. Il suo compito è quello di inviare il comando `DISCONNECT` al *Broker* usando esclusivamente l'*header* fisso costituito dai suoi due byte `0xE0` e `0x00`. Dopo l'invio di questo messaggio, il *socket* `mqtt-socket` viene chiuso e impostato a `0`.

- `mqtt-ping ( -- )`: anche questa parola implementa un comando molto semplice che è il `PINGREQ`. Anche qui è presente solo l'*header* fisso costituito dai byte `0xC0` e `0x00`.

- `mqtt-send ( msg topic -- )`: attraverso questa parola è possibile pubblicare sul *Broker* un messaggio `msg` per un particolare `topic`. Questa parola quindi implementa il comando `PUBLISH`.

  I due byte dell'*header* fisso, 0x30 e 0x02, identificano rispettivamente il comando e la lunghezza "attuale" del messaggio (questa parola è stata implementata per consentire la massima libertà di scrittura del *Topic* e del *payload* del messaggio. Per questo motivo le varie lunghezze non sono state fissate, ma sono modificate dinamicamente in base alle lunghezze del *Topic* e del *payload* del messaggio).

  L'*header* variabile prevede la lunghezza del *Topic* e il *Topic* stesso. Le due linee successive della parola si occuperanno di copiare il *Topic* all'interno del *buffer* e di determinare la sua lunghezza; la lunghezza, determinata dalla parola `strlen`, viene memorizzata all'interno del secondo byte dell'*header* variabile (quindi il quarto dell'intero messaggio). Le due linee successive della parola si occupano di copiare il *payload* del messaggio all'interno di `buffer` e di aggiornare la lunghezza totale del messaggio, contenuta nel secondo byte, sommando la lunghezza del *payload* e la lunghezza del *Topic*. Una volta preparato il messaggio e copiato all'interno di `buffer`, questo viene inviato al *Broker* tramite la parola `netcon-write-buf`.

- `mqttread ( -- length | -1 se disconesso )`: questa parola si occupa di leggere i messaggi inviati dal *Broker*. Ognuno di questi messaggi contiene nel byte 1 il codice del comando e nel byte 2 la lunghezza del messaggio. Questa parola legge questi due byte dal *socket* e li memorizza all'interno di `buffer`. Se la lunghezza del messaggio è maggiore di `0`, parte un ciclo `do loop` che memorizza all'interno di `buffer` i caratteri contenuti nel *socket*. Al termine, la lunghezza totale del messaggio viene inserita in cima al *Data Stack*. In caso di errore o connessione chiusa, la `netcon-read` restituisce il valore `-1` e, tramite le parole `unloop` ed `exit` si interrompe il ciclo `do loop`, inserendo il valore `-1` in cima al *Data Stack*.

- `mqttread-worker ( -- )`: questa parola viene usata come *task* da memorizzare all'interno di `mqttread-task` e si occupa di ricevere i messaggi ricevuti dal *Broker*. In questa versione i messaggi ricevuti vengono mostrati sull'*output stream*; come lavoro successivo, si dovrà aggiungere la parola `eval` per potere interpretare ed eseguire i messaggi contenenti codice

simbolico. Il *task* è implementato con un ciclo `begin while repeat` che esegue la parola `mqttread` per eseguire il *parsing* dei vari messaggi ricevuti dal *Broker*. Se il ciclo viene interrotto per qualsiasi motivo, la variabile `mqtt-socket` viene impostata a `0`.

- `mqtt-ping-worker ( -- )`: questa parola è il *task* che si occupa di eseguire ogni 30 secondi il comando `PINGREQ` tramite la parola `mqtt-ping`.
- `mqtt-receive ( topic -- )`: questa parola consente di effettuare la sottoscrizione ad un *Topic* `topic` inviando il comando `SUBSCRIBE`. I primi due byte, `0x82` e `0x04` sono quelli dell'*header* fisso. I due byte successivi, `0x00` e `0x0A`, indicano l'ID del messaggio. Successivamente inizia il payload del messaggio in cui, i primi due byte (temporaneamente impostati a `0x00`), contengono il numero di byte del *Topic* a cui ci si sta registrando. I byte successivi vengono riempiti con il contenuto di `topic`, aggiornando di conseguenza i byte che contengono le lunghezze.

  Successivamente, il messaggio contenuto in `buffer` viene inviato al *Broker* e si rimarrà in attesa del messaggio `SUBACK` (che richiede la lettura di 6 caratteri dal *socket*); al termine si avvia il *task* `mqttread-task` con la parola `mqttread-worker`.
- `mqtt`: questa parola memorizza l'inizio dell'array utilizzato per la memorizzazione degli *execution token* di `mqtt-connect mqtt-disconnect mqtt-send mqtt-receive`. Combinata con le parole `connect`, `disconnect`, `send` e `receive` (definite in *utils.forth*) consente l'esecuzione delle varie funzionalità precedentemente descritte.

6.6 Configurazione dei nodi

Come anticipato nella sezione 4.7.5, rispetto alle versioni precedenti, in Punyforth è possibile utilizzare delle parole che consentono la gestione dei moduli e l'accesso alla flash SPI direttamente usando il codice simbolico. Queste nuove funzionalità hanno dato spunto alla definizione e alla realizzazione di alcune parole attraverso le quali è possibile memorizzare, all'interno della flash SPI, stringhe di codice simbolico, rendendo possibile, nel contesto dell'IoT per la configurazione di dispositivi, la memorizzazione delle definizioni utili a definire la configurazione e il comportamento di quel particolare dispositivo, rendendo disponibile queste informazioni anche dopo il suo riavvio.

Per questo scopo, sono state utilizzate le parole `erase-flash`, `write-flash` e `read-flash` (viste nella sezione 4.7.5) attraverso le quali, all'interno del file *utils.forth* sono state definite le parole:

- `wp ( block -- )`: questa parola usa solamente la parola `erase-flash` e quindi si occupa di cancellare il contenuto di un blocco `block` di flash. Il codice di stato messo in cima al *Data Stack* dalla parola `erase-flash` attualmente viene ignorato e rimosso dal *Data Stack* tramite la parola `drop`.

  ```
  : wp ( block -- ) erase-flash drop ;
  ```

  Ad esempio, se si volesse cancellare il contenuto del blocco 256, si può eseguire `256 wp`.

- `wr ( data block -- )`: questa parola, usando la parola `write-flash`, scrive il contenuto della stringa `data` all'interno del blocco `block`.

  Come descritto nella sezione 4.7.5, prima di potere eseguire la scrittura di un blocco, è necessario effettuare la sua cancellazione; questa parola quindi, per prima cosa, esegue la cancellazione del blocco `block` tramite la parola `wp`. Le parole successive si occupano di preparare il *Data Stack* con tutto ciò che serve alla parola `write-flash`. Si ricorda che la parola `write-flash` si aspetta di trovare all'interno del *Data Stack* i dati `address size buffer`; questi parametri si ottengono nel seguente modo: la parola `addr` ottiene l'indirizzo dell'inizio del blocco di flash SPI moltiplicando `block` per la costante `SIZE`; le parola `strlen` ottiene la lunghezza della stringa contenuta all'interno di `data`, contando i caratteri fino al carattere terminatore di stringa `NUL`; le operazioni di `swap` e `rot` sul *Data Stack* fanno si che i dati siano posizionati secondo l'ordine corretto. Il codice di stato inserito in cima al *Data Stack* dalla parola `write-flash` attualmente viene ignorato e rimosso dal *Data Stack* tramite la parola `drop`.

  Se si volesse scrivere all'interno del blocco 256 la stringa "`: ON HIGH ;`" che definisce la parola `ON` si può eseguire `": ON HIGH ;" 256 wr`.

- `rd ( size buffer address -- )`: questa parola usa solamente la parola `read-flash` e quindi si occupa di leggere `size` caratteri dalla flash SPI a partire da `address` copiandoli all'interno di `buffer`. Il codice di stato messo in cima al *Data Stack* dalla parola `read-flash` attualmente viene ignorato e rimosso dal *Data Stack* tramite la parola `drop`.

Se si volesse leggere il contenuto del blocco 256 scrivendolo all'interno di un *buffer* si può eseguire `SIZE data 256 addr rd`.

- `>module ( buffer -- )`: questa parola converte il codice simbolico contenuto in `buffer` (il cui indirizzo si trova in cima al *Data Stack*) in un modulo per il caricamento in flash. Semplicemente, questa parola aggiunge i caratteri `\r\n/end\r\n\r\n` alla fine di `buffer`.
- `conf! ( buffer -- )`: usa la parola `>module` per convertire `buffer` in un modulo. Terminata la conversione, usa la parola `wr` per scrivere il contenuto di `buffer` all'interno del blocco `CONF`.

All'interno del file *utils.forth* (consultabile alla sezione 9.2.12) si è definita la costante `CONF` (sul blocco 256) dove sarà caricata la configurazione come codice simbolico.

Si sono definite anche delle parole utili al caricamento della configurazione all'avvio della ESP8266:

- `ready? ( block -- block bool )`: questa parola controlla se il blocco `block` contiene nel primo byte il valore `255` che identifica che il blocco di flash SPI non è stato scritto e non contiene codice simbolico. Se il valore è uguale a 255, la parola inserisce in cima la *Data Stack* il valore 0, altrimenti inserisce -1 per indicare che il blocco contiene una configurazione.
- `?load ( block bool -- )`: questa parola esegue la parola `load` sul blocco `block` se il valore di `bool` è vero.

Il file main.forth (consultabile alla sezione 9.2.13), contiene le istruzioni in codice simbolico che vengono eseguite all'avvio di Punyforth. Tra queste, in merito alla configurazione, si trova la linea

```
CONF ready? ?load
```

che prova a caricare il contenuto del blocco `CONF` se il blocco contiene una configurazione valida.

7 Valutazione Sperimentale

L'attività di test è stata svolta utilizzando l'MCU ESP8266-12E saldata sulla scheda di sviluppo NodeMCU AMICA, le cui descrizioni sono state fornite rispettivamente nel capitolo 3 e nella sezione 5.1. Si sono utilizzati dei LED come attuatori e un sensore di luminosità analogico GL5516.

7.1 Memoria

Purtroppo riguardo alla memoria utilizzata da Punyforth in flash SPI e in RAM sulla ESP8266 non si è trovata documentazione, probabilmente perché l'occupazione di queste aree dipende esclusivamente dalla dimensione dell'immagine binaria e dalla quantità di moduli installati e caricati. Si sono condotti dei test a riguardo per valutare l'occupazione di memoria in flash e in RAM da ciò che è stato utilizzato nell'attività progettuale.

Utilizzando il comando Python dalla *shell* di Mac OSX

```
python flash.py /dev/cu.SLAB_USBtoUART --modules CORE FLASH TASKS NETCON WIFI TCPREPL MAILBOX RINGBUF GPIO UTILS TCPCLI MQTTCLI --main main.forth --flashmode dio
```

si è provveduto ad eseguire il caricamento dei file binari e dei sorgenti *.forth* all'interno della flash SPI.

Elemento	Indirizzo	Dimensione Occupata (byte)
rboot.bin	0x0	4096
blank_config.bin	0x1000	2048
punyforth.bin	0x2000	321536
main.forth	0x52000	1024
Moduli .forth	0x53000	38912

Tabella 10 - Occupazione memoria flash SPI

Dall'output del comando precedente è stato possibile ricostruire l'occupazione della memoria flash SPI (i dati indicati in tabella comprendono il *padding* per raggiungere il multiplo di 1024 byte più vicino) e costruire la Tabella 10. Lo spazio totale occupato all'interno della flash SPI contenente i file binari e i moduli specificati all'interno del comando precedente è quindi di 367616 byte.

Per quanto riguarda la RAM, si ricorda dalla sezione 3.2 che la ESP8266 ha circa 50 KB disponibili una volta che il sistema operativo è avviato. Attraverso l'uso delle parole `osfreemem`, `usedmem` e `freemem` di Punyforth è stato possibile misurare la RAM occupata e disponibile.

Parola	Risultato (byte)
osfreemem	14372
usedmem	30156
freemem	5028

Tabella 11 - Memoria RAM occupata e disponibile

Dalla Tabella 11 si può vedere il risultato delle parole eseguite. La parola `osfreemem` restituisce il quantitativo di RAM ancora disponibile per il sistema operativo. Si ricorda dalla sezione 5.4 che il dizionario di Punyforth è inserito all'interno dell'*heap*; le parole `usedmem` e `freemem` misurano la quantità di RAM rispettivamente occupata e libera all'interno dell'*heap* riservato a Punyforth e specificato all'interno del file *data.S*. La somma dei valori di `usedmem` e `freemem` in Tabella 11 risulta 35184 byte e rappresenta lo spazio totale dell'*heap* riservato a Punyforth. Si può vedere che lo spazio rimanente per le nuove definizioni è di 5028 byte (a seguito del caricamento all'interno del dizionario di tutti i moduli memorizzati in flash tramite il comando `python flash.py` precedente). Ogni nuova definizione va ad occupare questa porzione di *heap* e al suo esaurimento la ESP8266 eseguirà un riavvio per esaurimento di memoria disponibile. Punyforth non contiene nessun meccanismo di *garbage collection* per cui, per usi che richiedono diverse definizioni eseguite nel tempo, il riavvio descritto in precedenza sarà inevitabile.

7.2 Dimensione dell'immagine binaria e dei sorgenti

Dopo aver valutato l'occupazione in flash, si è provveduto ad analizzare l'occupazione dell'immagine binaria e dei sorgenti del progetto (Figura 18).

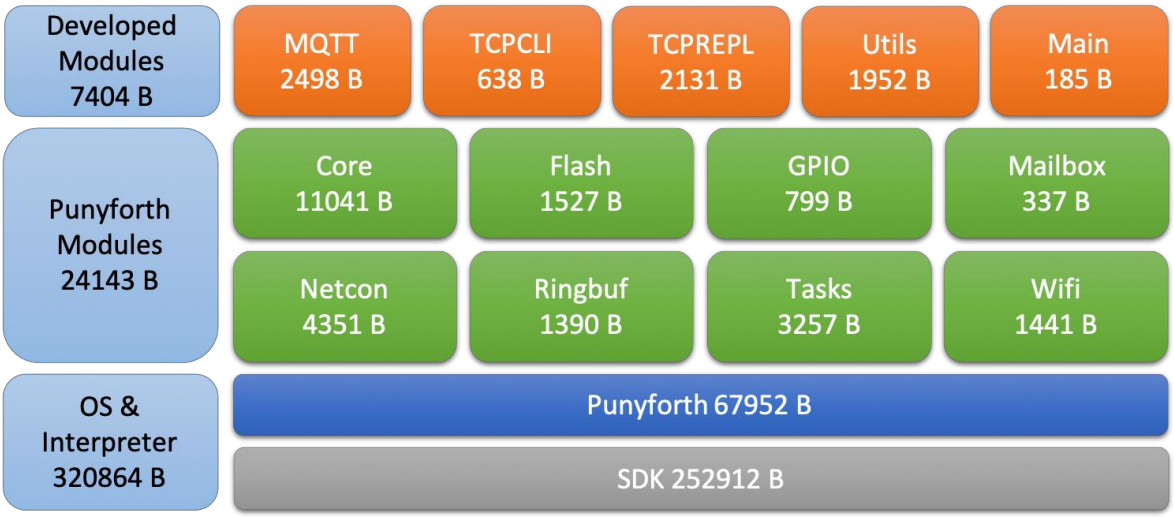

Figura 18 - Dimensioni immagini binarie e moduli

7.2.1 Immagine binaria del Sistema Operativo con l'SDK

Innanzitutto, si è calcolata la dimensione dell'immagine binaria contenente il solo Sistema Operativo e l'SDK. Per ottenere questa misura, si è provveduto a compilare l'immagine binaria scrivendo un programma per la ESP8266 che esegue un *loop* vuoto; all'interno di questo piccolo programma sono state usate le librerie dell'SDK (tramite la direttiva `#include`). La compilazione è avvenuta su un *container Docker* basato sull'immagine di *Debian* su cui sono stati installati i sorgenti dell'SDK e tutti gli strumenti utili per la compilazione all'interno delle cartelle */home/iot/esp/esp-open-rtos* e */home/iot/esp/esp-open-sdk* come indicato nella sezione 5.5. Esattamente si sono svolti i seguenti passi:

All'interno della cartella */home/iot/esp/esp-open-rtos/examples* è stata creata la cartella *sdk*. All'interno della cartella *sdk* sono stati creati due file:

- *user_main.c*: contiene il programma con il *loop* vuoto e le librerie per includere l'SDK.
- *Makefile*: contiene le direttive per la compilazione.

Si è lanciato il comando `export PATH=$PATH:~/esp/esp-open-sdk/xtensa-lx106-elf/bin` per rendere disponibile alla sessione di *shell* attiva il compilatore per l'architettura *Xtensa*.

Si è lanciato il comando `make` per eseguire la compilazione dell'immagine binaria. L'immagine binaria compilata si trova posizionata in */home/iot/esp/esp-open-rtos/examples/sdk/firmware/SDK.bin*. Dalla *shell* è bastato eseguire il comando `ls -l` per ottenere la dimensione in byte del file contenente l'immagine binaria dell'SDK. La dimensione ottenuta è stata di 252912 byte.

7.2.2 Immagine binaria di Punyforth

La dimensione dell'immagine di Punyforth è stata ottenuta eseguendo il comando `ls -l`. La dimensione ottenuta è stata di 320864 byte. Effettuando la differenza tra la dimensione dell'immagine binaria di Punyforth e quella dell'immagine contenente soltanto l'SDK, si è ottenuta la reale dimensione occupata da Punyforth che è di 67952 byte (vedi Tabella 12).

Immagine Binaria	Dimensione (byte)
OS SDK	252912
Punyforth	320864
Delta	67952

Tabella 12 - Dimensioni delle immagini binarie

7.2.3 Moduli scritti in Codice Simbolico

Si sono misurate anche le dimensioni dei sorgenti dei moduli scritti in codice simbolico, differenziando quelli di Punyforth da quelli scritti per l'attività progettuale. La misura è stata effettuata usando l'output del comando `ls -l`. Il risultato è visibile in Tabella 13.

Tipologia Moduli	Dimensione (byte)
Punyforth	24143
Sviluppati	7404
Totale	31547

Tabella 13 - Dimensione moduli scritti in Codice Simbolico

7.3 Test di Configurazione

Il test di configurazione è stato condotto collegando, ai PIN *header* della scheda di sviluppo NodeMCU AMICA le seguenti componenti (Figura 19):

- due LED (uno rosso e uno verde) ai PIN GPIO 12 e 14.
- un sensore di luminosità (gl5516) al PIN ADC.

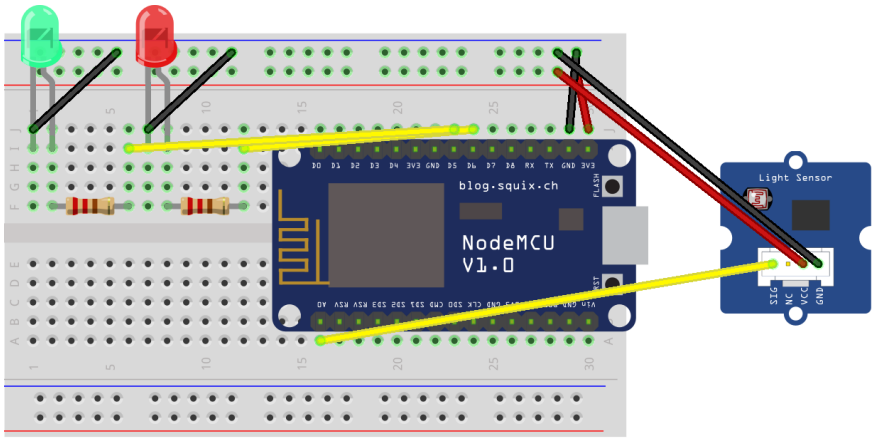

Figura 19 - Collegamenti LED e sensore su myesp

All'interno della base di conoscenza si è definita un'istanza dell'MCU ESP8266 chiamata `myesp` di tipo `esp8266_12e` tramite

```
mcu_name(myesp, esp8266_12e).
```

indicando il suo indirizzo IP e la porta del server TCP REPL tramite

```
mcu_net_address(myesp, 'myesp.local', 1983).
```

A questo punto, la base di conoscenza è stata arricchita con tutte le informazioni necessarie per poter generare il codice simbolico rappresentante la configurazione (si veda la sezione 9.3.1 per il codice risultante).

Usando le regole

```
mcu_gen_conf_code(myesp, McuBaseConf),
forth_mcu_gen_conf_code(McuName, McuConf),
flatten([McuBaseConf, McuConf, "store"], Messages),
mcu_send_message(myesp, Messages).
```

il sistema di regole si è collegato al server TCP *REPL* di `myesp` inviando e memorizzando la configurazione all'interno del blocco `CONF` nella flash SPI (schema visibile in Figura 20).

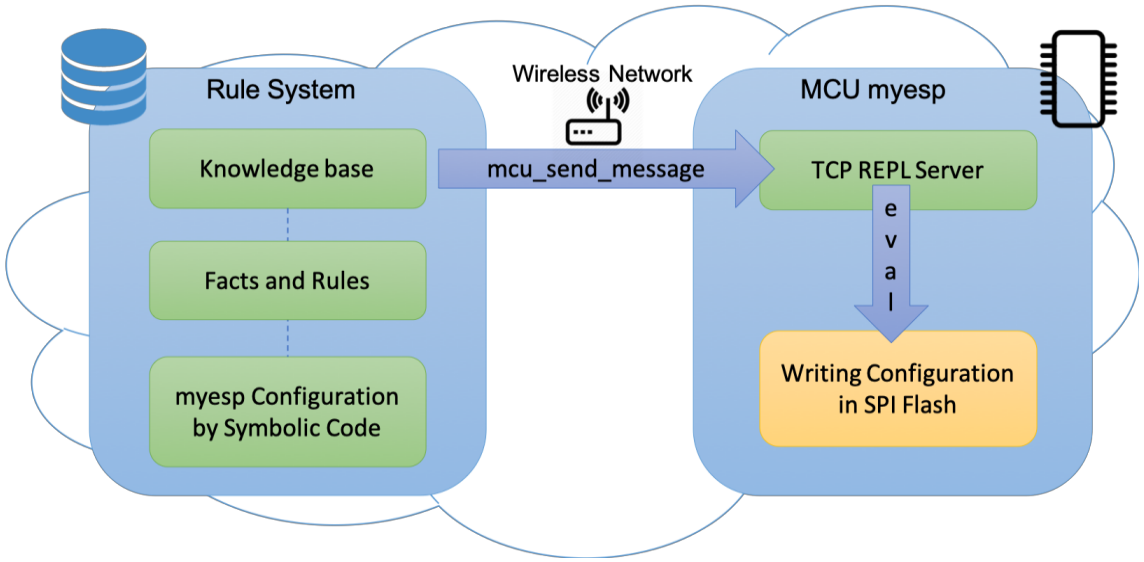

Figura 20 - Invio, ricezione e memorizzazione della configurazione

In questo modo, al riavvio dell'MCU, Punyforth ha eseguito la linea

```
CONF ready? ?load
```

caricando la configurazione memorizzata e rendendo disponibili le definizioni caricate per il loro utilizzo.

Infatti, usando la regola

```
mcu_send_message(myesp, ['RLED ON 500 ms RLED OFF\r\n']).
```

è stato possibile accendere il LED rosso per poi spegnerlo dopo 500 millisecondi (schema visibile in Figura 21).

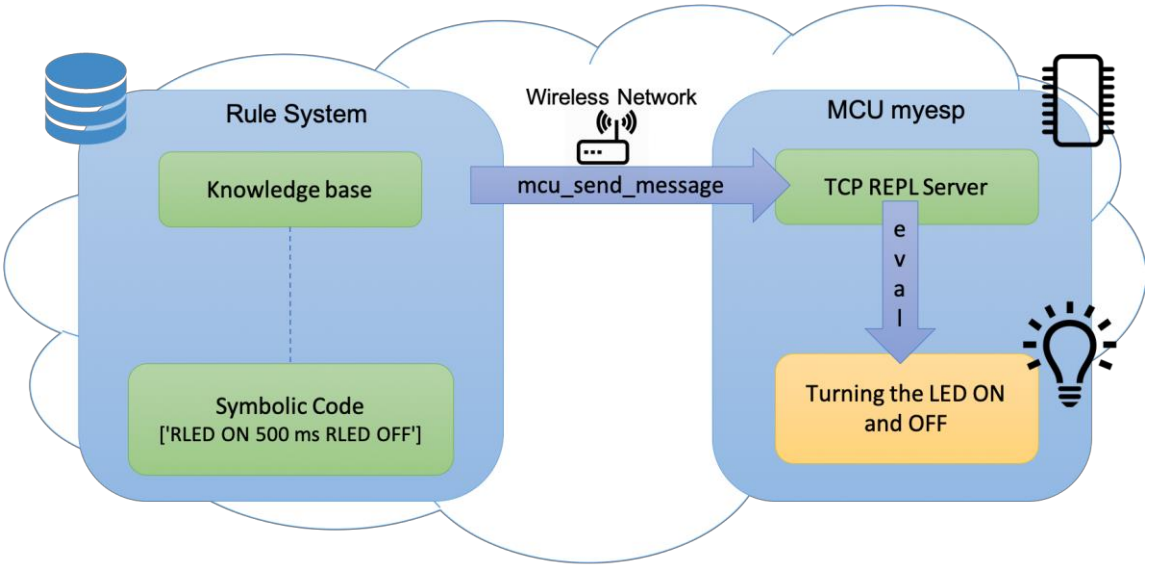

Figura 21 - Esecuzione azione al riavvio

7.4 Tempo di trasmissione del codice simbolico

Per calcolare tempo impiegato per il trasferimento del codice simbolico da un client TCP al server TCP *REPL* dell'MCU, si è provveduto a scrivere un client TCP in Python. Il codice simbolico da inviare all'MCU tramite TCP è stato memorizzato su un file che viene aperto dallo script Python. I file interessati per effettuare questa misura sono stati inseriti all'interno della cartella *punyforth/arch/esp8266/bin* del progetto e sono:

- *client.py*: contiene lo script Python del client TCP (si veda la sezione 9.3.2).
- *test_conf.forth*: contiene il codice simbolico da trasmettere all'MCU tramite TCP (il codice simbolico trasmesso è quello risultante dal processo di configurazione visto nella sezione 7.3 e consultabile nella sezione 9.3.1).

Il test effettuato è stato quello di inviare per 100 volte la stessa quantità di codice simbolico all'MCU, misurando interamente e per ogni tentativo il tempo del seguente processo:

- Connessione
- Invio, linea per linea, del codice simbolico
- Memorizzazione del codice simbolico in flash
- Disconnessione

Sono state fatte 4 prove di trasmissione di codice simbolico, raddoppiando, di volta in volta, la quantità di byte da inviare all'MCU. In totale, quindi, sono state eseguite 400 sessioni di trasmissione di codice simbolico. I dati raccolti sono stati aggregati e raggruppati per quantità di byte inviati (usando un foglio di calcolo) misurando il valore minimo, il valore medio e il valore massimo dei tempi di trasmissione ottenuti (in secondi). I dati raccolti sono visibili in Tabella 14 e in Figura 22.

Byte Trasmessi	Min (secondi)	Media (secondi)	Max (secondi)
352	0,057426	0,19377499	0,522646
697	0,077489	0,22295915	0,532557
1387	0,104296	0,19822329	0,555743
2767	0,150337	0,19028826	0,586235

Tabella 14 - Valore minimo, medio e massimo dei tempi di trasmissione per byte trasmessi

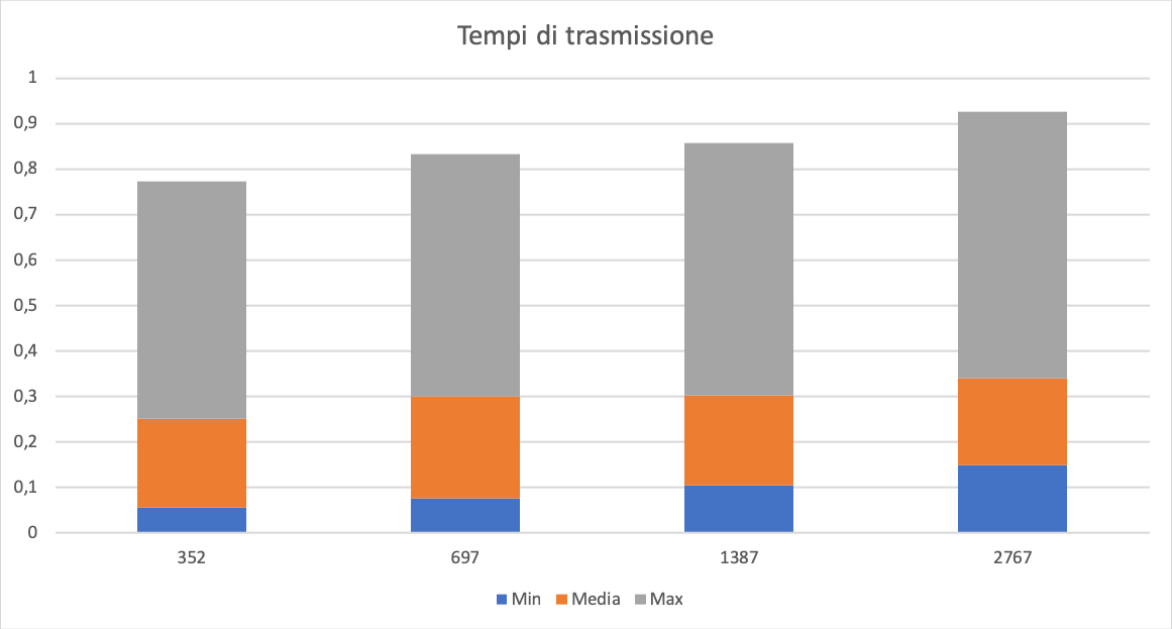

Figura 22 - Grafico dei tempi di trasmissione di codice simbolico

In dettaglio, lo script Python, utilizzato per simulare il client TCP, esegue le seguenti operazioni:

1. Apre il file contenente il codice simbolico da trasmettere, memorizzando le linee di codice simbolico all'interno di una lista.
2. Esegue un loop per 100 volte. All'interno del loop:
    a. Memorizza in `t1` il `datetime` corrente.
    b. Apre una connessione verso l'MCU.
    c. Invia le linee di codice simbolico contenute all'interno della lista.
    d. Invia la stringa *store* per indicare all'MCU che deve memorizzare il codice simbolico contenuto all'interno del buffer.
    e. Invia la stringa *quit* per chiudere la connessione e chiude la connessione.
    f. Memorizza in `t2` il `datetime` corrente
    g. Calcola il `delta_t` tra `t2` e `t1` e lo stampa sullo standard output della shell locale per la consultazione successiva.

Il client che ha eseguito la trasmissione dei dati e l'MCU sono stati connessi allo stesso *Access Point* tramite la rete Wi-Fi.

7.5 Comunicazione tra nodi tramite TCP REPL e TCP Client

Questo test è stato condotto usando due ESP8266-12E: la prima (chiamata "esp1" con indirizzo IP "192.168.4.2") è stata configurata collegando un sensore di luminosità GL5516 al PIN ADC; la seconda (chiamata "esp2" con indirizzo IP "192.168.4.5") è stata configurata collegando un LED giallo al PIN 12 (vedi Figura 23). Entrambi gli MCU sono stati collegati allo stesso *Access Point*.

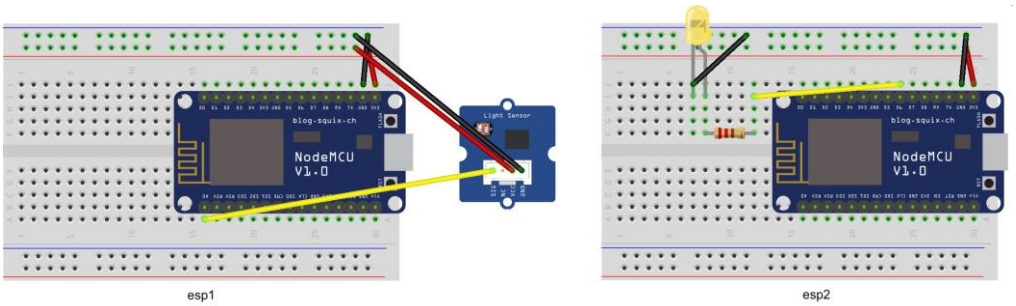

Figura 23 - Collegamenti di esp1 e esp2

L'idea che sta alla base di questo test è quello di fare comunicare i due MCU tramite *socket* TCP (attraverso la rete Wi-Fi) usando codice simbolico; il comportamento desiderato è il seguente:

1. "esp2" chiede a "esp1" di interrogare il sensore di luminosità e di ricevere il risultato.
2. "esp2", appena riceve la risposta, lo confronta con un valore di soglia (200) e se il valore misurato è maggiore del valore di soglia (assenza di luce) accende il proprio LED giallo, in caso contrario (presenza di luce) spegne il LED giallo.

All'interno della configurazione dei due MCU, si sono effettuate le seguenti definizioni (usando codice simbolico) per identificare il sensore e l'attuatore collegati:

- esp1: `: LUMSENS nop ;` (`LUMSENS` identifica il sensore di luminosità).
- esp2: `12 CONSTANT YLED` (`YLED` identifica il LED giallo).

All'interno della configurazione di "esp2" si è provveduto a definire il comportamento illustrato nei punti precedenti, utilizzando il codice simbolico presente all'interno della sezione 9.3.3; si sono definite le parole:

- `check ( n -- )`: consuma l'interno `n` dal *Data Stack* per confrontarlo con il valore di soglia impostato ed eseguire l'accensione o lo spegnimento di `YLED`.

- `test`: definisce il *task* che si occupa di interrogare "esp1". Consiste in un ciclo `begin while repeat` la cui esecuzione continua finché il contenuto della variabile `test-running` è `-1`. Il suo compito è quello di inviare il codice simbolico

  ```
  LUMSENS READ . tcp-response return
  ```

  e poi mettersi in attesa sulla *mailbox* `tcpcli-mailbox` (di cui si è parlato all'interno della sezione 6.4). Nel momento in cui "esp2" risponde, il *task* viene sbloccato perché riceve l'intero (rappresentante il valore di luminosità misurato dal sensore) all'interno della *mailbox* ed esegue l'azione di verifica del valore ricevuto tramite la parola `check` che, a sua volta, esegue l'azione di accensione o spegnimento di `YLED`; terminata l'azione, procede ad interrogare nuovamente "esp2" tramite il codice simbolico visto in precedenza, ricominciando nuovamente il ciclo.
- `test-start`: connette "esp2" a "esp1" (tramite `tcp connect`), imposta la variabile `test-running` a `-1` e avvia il *task* `test-task` usando la parola `test`.
- `test-stop`: disconnette "esp2" da "esp1" (tramite `tcp disconnect`) e imposta la variabile `test-running` a `0` per indicare al ciclo `begin while repeat` di `test` che deve interrompersi.

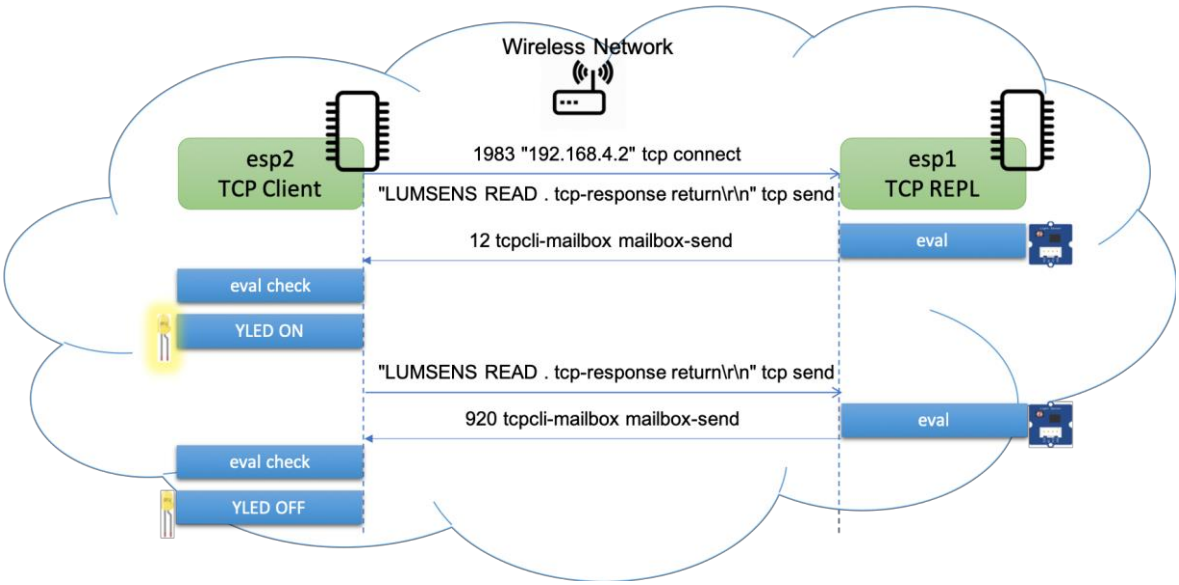

Figura 24 - Comunicazione tra nodi tramite TCP REPL e TCP Client

Il test è stato svolto con esito positivo riscontrando il funzionamento previsto senza problemi. Lo scambio del codice simbolico è mostrato in Figura 24.

7.6 Client MQTT

Questo test è stato condotto usando due ESP8266-12E: la prima (chiamata "esp1" con indirizzo IP "192.168.0.14") è stata configurata collegando un LED rosso al PIN 12; la seconda (chiamata "esp2" con indirizzo IP "192.168.0.15") è stata configurata collegando un LED verde al PIN 12. Su entrambi gli MCU si è provveduto ad inserire nella configurazione la costante

```
12 constant: LED
```

Entrambi gli MCU sono stati utilizzati come *Subscriber* per ricevere le azioni da eseguire inviate su dei *Topic* dedicati. In questo test, è stato usato come *Broker* un server Fedora 27 (chiamato "vls1" con indirizzo IP "192.168.0.19") con l'applicazione "mosquitto" in esecuzione. Sulla stessa macchina si è eseguito un *Publisher* tramite l'applicazione "mosquitto_pub".

Tramite un terminale *Telnet* si è effettuata la connessione al server TCP *REPL* di "esp1" e si sono inviate le seguenti stringhe per collegarsi al *Broker* e registrarsi sul *Topic* dedicato

```
1883 "192.168.0.19" mqtt connect
"casa/cucina/tavolo/led/red" mqtt receive
```

La stessa cosa si è fatta per la "esp2" inviando le stringhe

```
1883 "192.168.0.19" mqtt connect
"casa/cucina/tavolo/led/green" mqtt receive
```

Dalla shell del *Publisher* si è eseguito

```
mosquitto_pub -h 192.168.0.19 -p 1883 -t
"casa/cucina/tavolo/led/#" -m "LED ON"
```

per inviare a tutti i `led` presenti sul `tavolo` in `cucina` il messaggio `LED ON`; il *Broker* ha provveduto ad inviare il messaggio ai due MCU registrati; alla ricezione del messaggio sul *socket*, sull'*output buffer* di entrami gli MCU sono stati mostrati i caratteri del messaggio (si ricorda dalla sezione 6.5.1 che l'attuale implementazione del *Subscriber* non prevede l'`eval` della stringa ricevuta). Lo scambio del codice simbolico tramite protocollo MQTT è schematizzato in Figura 25.

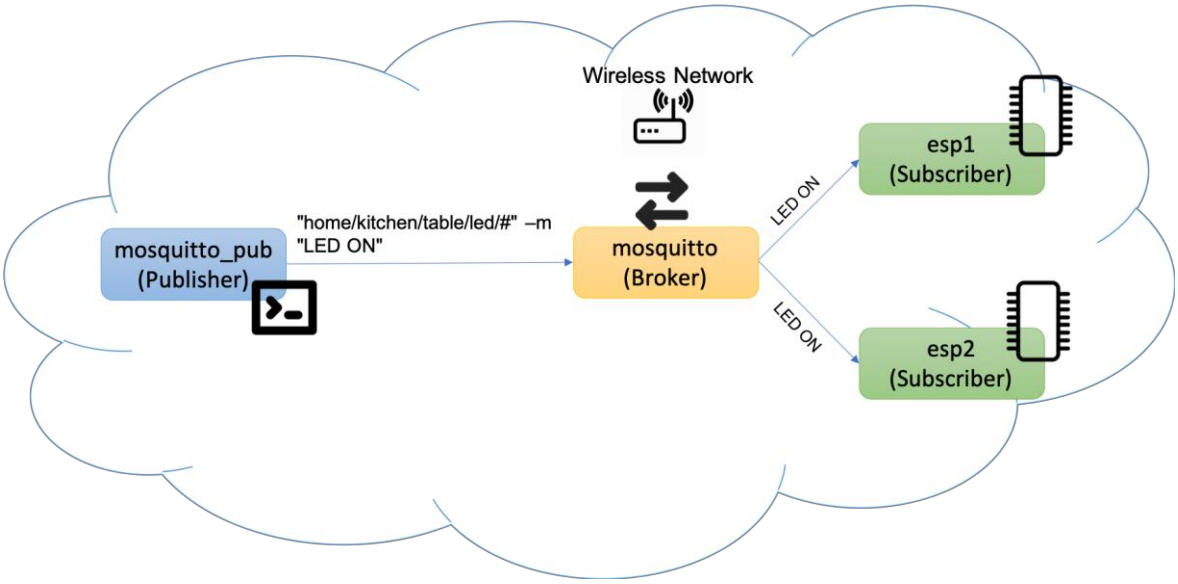

Figura 25 - Test di invio e ricezione di codice simbolico tramite protocollo MQTT

7.6.1 Confronto dimensione dei sorgenti Client MQTT e ESP-RTOS-MQTT

Sempre in merito al client *MQTT* si è voluto eseguire un confronto tra l'implementazione eseguita all'interno dell'attività progettuale usando il linguaggio FORTH e l'implementazione presente sul Web *ESP-RTOS-MQTT* usando il linguaggio C. Il criterio con cui si è eseguita la misurazione è stato il numero di righe di codice e il numero di caratteri utilizzati. Usando dalla *shell* di MAC OSX il comando

```
find . -name '*.c' -o -name '*.h' | xargs wc -lc
```

si è effettuato il conteggio delle righe di codice e dei caratteri di *ESP-RTOS-MQTT*; il risultato è stato di 3220 righe e di 97006 caratteri in totale tra codice e commenti. La stessa operazione è stata fatta sull'implementazione eseguita in FORTH usando il comando

```
wc -lc mqtt-client.forth
```

risultando avere 132 righe e 2420 caratteri in totale tra codice commenti. I valori sono riportati in Tabella 15 e in Figura 26.

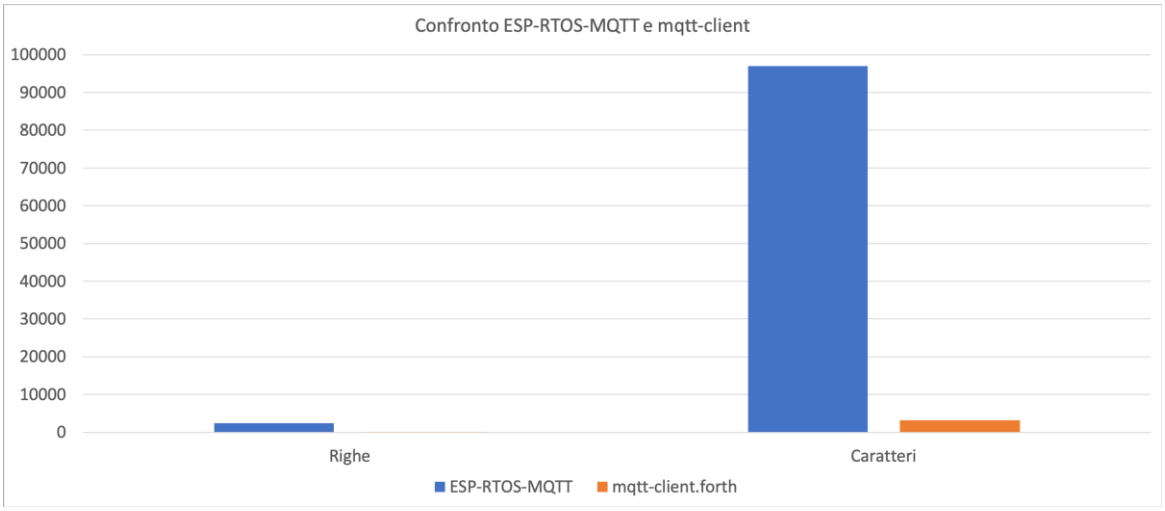

Figura 26 - Grafico confronto ESP-RTOS-MQTT e mqtt-client.forth

Implementazione	Righe	Caratteri
ESP-RTOS-MQTT	3220	97006
mqtt-client.forth	132	2420

Tabella 15 - Risultato confronto ESP-RTOS-MQTT e mqtt-client.forth

8 Conclusioni e possibili sviluppi

Dai test effettuati si conferma la possibilità di utilizzare delle stringhe di codice simbolico per potere effettuare la configurazione e la programmazione di dispositivi con risorse limitate in modo semplice ed efficiente; tramite questo sistema è possibile effettuare queste operazioni in tempo reale e da remoto tramite l'utilizzo di reti TCP/IP.

Come si è visto dai test, l'ambiente simbolico utilizzato (con in aggiunta le implementazioni svolte per l'attività progettuale) ha un'occupazione ragionevole di memoria flash e RAM.

Il linguaggio FORTH utilizzato consente una facile espandibilità senza la necessità di dichiarare costrutti complessi come classi, metodi e funzioni tipici di altri linguaggi di programmazione di livello più basso, consentendo l'espansione delle funzionalità dei i nodi della rete in base alle proprie esigenze. Inoltre, si è potuto verificare l'alto livello di compressione che ha questo linguaggio rispetto ad altri per potere eseguire la programmazione di nuove funzionalità: con poche righe di codice si possono implementare diverse funzionalità; il suo linguaggio ad alto livello consente anche un alto grado di espressività, aprendo la strada al diretto utilizzo di un linguaggio naturale per l'esecuzione delle operazioni sui nodi.

L'implementazione dei protocolli di comunicazione (*Client TCP* e *Client MQTT*) ha dato la possibilità ai nodi di potere interagire con diversi altri dispositivi all'interno della rete, favorendo lo scambio di informazioni e l'interoperabilità per il raggiungimento di obbiettivi comuni.

L'interazione tra un sistema di regole e l'ambiente simbolico all'interno dei nodi della rete, dà all'utente (guidato da un sistema intelligente) la possibilità di utilizzare strumenti ad alto livello per interagire a basso livello con l'hardware dei vari nodi.

Da questa esperienza progettuale si sono rivelati però dei limiti su Punyforth; Punyforth è un'applicazione, non è un sistema operativo: se fosse un sistema operativo, probabilmente si potrebbe ridurre la dimensione dell'immagine binaria e la dimensione della RAM occupata; lo spazio riservato al dizionario risiede esclusivamente in RAM: il poco spazio rimanente può rapidamente consumarsi e non essere recuperabile poiché non esiste nessun meccanismo di *garbage collection*.

Individuati questi limiti, si potrebbe pensare di implementare un meccanismo di *garbage collection* per riutilizzare aree di memoria RAM non più utilizzate. Altra cosa che si potrebbe implementare è la possibilità di memorizzare il dizionario all'interno della memoria flash SPI, in modo da avere più memoria RAM libera.

Per quanto riguarda il protocollo *MQTT*, il client andrebbe migliorato implementando tutti i comandi esistenti. Si potrebbe implementare direttamente un *Broker* all'interno dei nodi della rete, rendendoli ancora più indipendenti da strutture hardware più complesse è più costose.

9 Appendice: Codice Sorgente

9.1 Punyforth

9.1.1 wifi.forth

```
: wifi-connect ( password ssid  -- | throws:EWIFI )
     STATION_MODE wifi-set-mode check-status
     wifi-set-station-config check-status
     wifi-station-connect check-status
;


: wifi-softap ( max-connections channels hidden authmode
password ssid -- | throws:EWIFI )
     SOFTAP_MODE wifi-set-mode check-status
     wifi-set-softap-config check-status
;
```

9.1.2 netcon.forth

```
: netcon-new ( type -- netcon | throws:ENETCON )
     override
     netcon-new dup 0= if ENETCON throw then
     RECV_TIMEOUT_MSEC over netcon-set-recvtimeout
;


: netcon-connect ( port host type -- netcon | throws:ENETCON
)
     override
     netcon-new dup >r netcon-connect check r>
;


: netcon-bind ( port host netcon -- | throws:ENETCON )
     override
     netcon-bind check
;
```

```
: netcon-listen ( netcon -- | throws:ENETCON )
     override
     netcon-listen check
;



: netcon-tcp-server ( port host -- netcon | throws:ENETCON )
     TCP netcon-new
     ['] netcon-bind keep
     dup netcon-listen
;


: netcon-udp-server ( port host -- netcon | throws:ENETCON )
     UDP netcon-new
     ['] netcon-bind keep
;


: netcon-accept ( netcon -- new-netcon | throws:ENETCON)
override
     begin
          pause
          dup netcon-accept dup NC_ERR_TIMEOUT <> if
               check nip
               RECV_TIMEOUT_MSEC over netcon-set-recvtimeout
               exit
          then
          2drop
     again
;


: netcon-send-buf ( netcon buffer len -- | throws:ENETCON )
     swap rot netcon-send check
;
```

```
: netcon-write-buf ( netcon buffer len -- | throws:ENETCON )
     swap rot netcon-write check
;


: netcon-write ( netcon str -- | throws:ENETCON ) override
     dup strlen netcon-write-buf
;


: netcon-writeln ( netcon str -- | throws:ENETCON )
     over swap netcon-write "\r\n" netcon-write
;


: read-ungreedy ( size buffer netcon -- count code |
throws:ERTIMEOUT )
     ms@ >r
     begin
          3dup netcon-recvinto
          dup NC_ERR_TIMEOUT <> if
               rot drop rot drop rot drop
               r> drop ( start time )
               exit
          else
               pause
          then
          2drop ( count code )
          dup netcon-read-timeout@ 0> if
               ms@ r@ - over netcon-read-timeout@ 1000 * > if
               ERTIMEOUT throw
               then
          then
     again
;
```

```
: netcon-read ( netcon size buffer -- count | -1 |
throws:ENETCON/ERTIMEOUT )
     rot read-ungreedy dup NC_ERR_CLSD = if
          2drop -1 exit
     then
     check
;


: netcon-readln ( netcon size buffer -- count | -1 |
throws:ENETCON/EOVERFLOW/ERTIMEOUT )
     swap 0 do
          2dup 1 swap i + netcon-read -1 = if
               i + 0 swap c! drop
               i 0= if -1 else i then
               unloop exit
          then
          dup i + c@ 10 = i 1 >= and if
               dup i + 1- c@ 13 = if
                    i + 1- 0 swap c! drop i 1-
                    unloop exit
               then
          then
     loop
     EOVERFLOW throw
;


: netcon-dispose ( netcon -- )
     dup netcon-close netcon-delete
;
```

9.1.3 task.forth

```
struct
    cell field: .next \ Puntatore al task successivo
    cell field: .status \ Stato del task
    cell field: .sp \ Stack Pointer del task
    cell field: .rp \ Return Stack Pointer del task
    cell field: .ip \ Instruction Pointer del task
    cell field: .s0 \ Indirizzo del TOS del Data Stack
    cell field: .r0 \ Indirizzo del TOS del Return Stack
    cell field: .handler \ Handler delle eccezioni
constant: Task


: task: ( user-space-size "name" ) ( -- task )
    create:
        here
        swap Task + allot
        SKIPPED             over .status !
        last @ .next @      over .next !
        dup last @               .next !
        alloc-stack         over .sp !
        alloc-rstack        over .rp !
        0                   over .handler !
        dup .sp @ over .s0 !
        dup .rp @ over .r0 !
        last !
;


: activate ( task -- )
    r> over .ip !
    PAUSED current @ .status !
    sp@ cell + r> rp@ save
    switch
;
```

```
: stop ( task -- )
     SKIPPED swap .status ! choose switch
;


: deactivate ( -- )
     current @ stop
;
```

9.1.4 mailbox.forth

```
: mailbox: ( size ) ( -- mailbox ) ringbuf: ;


: mailbox-send ( message mailbox -- )
    begin
         dup ringbuf-full?
     while
          pause
     repeat
     ringbuf-enqueue
;

: mailbox-receive ( mailbox -- message )
     begin
          dup ringbuf-empty?
     while
          pause
     repeat
     ringbuf-dequeue
;
```

9.1.5 tcp-repl.forth

```
: server ( task -- )
    activate
    PORT HOST netcon-tcp-server
    begin
        print: 'PunyREPL started on port ' PORT .
        print: ' on host ' HOST type cr
        dup netcon-accept
        connections mailbox-send
    again
    deactivate
;


: command-loop ( -- )
    begin
        client @ 128 line netcon-readln -1 <>
        line "quit" =str invert and
    while
        line strlen if line eval then
    repeat
;


: worker ( task -- )
    activate
    begin
        connections mailbox-receive client !
        ['] command-loop catch ?dup if
            ex-type
        then
        client @ netcon-dispose
        0 client !
    again
    deactivate
;
```

```
: eval ( str -- i*x )
    0 #tib !
    tib >in !
    dup strlen 0 do
        dup i + c@ chr>in
    loop
    13 chr>in 10 chr>in
    drop
    push-enter
;


: repl-start ( -- )
    println: 'Starting PunyREPL..'
    multi
    ['] type-composite xtype !
    ['] emit-composite xemit !
    repl-server-task server
    repl-worker-task worker
;
```

9.1.6 macros.S

```
defprimitive "funzione",8,funzione,REGULAR
    DPOP a3
    DPOP a2
    CCALL funzione_scritta_in_linguaggio_c
    DPUSH a2
    NEXT
```

9.2 Implementazione

9.2.1 aes.c

```
#include <stdint.h>
#include <string.h> // CBC mode, for memset
#include "aes.h"

#define Nb 4
#define BLOCKLEN 16 //Block length in bytes AES is 128b block only

#define Nk 4        // The number of 32 bit words in a key.
#define KEYLEN 16   // Key length in bytes
#define Nr 10       // The number of rounds in AES Cipher.
#define keyExpSize 176
#define OUTBUFFERSIZE 136

// state - array holding the intermediate results during decryption.
typedef uint8_t state_t[4][4];
static state_t *state;

// The array that stores the round keys.
static uint8_t RoundKey[keyExpSize];

// The Key input to the AES Program
static const char *Key;

// Initial Vector used only for CBC mode
static char *Iv;
```

```
static char IV[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0};


// The lookup-tables are marked const so they can be placed in
read-only storage instead of RAM

// The numbers below can be computed dynamically trading ROM
for RAM -

// This can be useful in (embedded) bootloader applications,
where ROM is often limited.

static const uint8_t sbox[256] = {

        //0     1    2      3     4    5     6     7      8
9     A      B    C     D     E     F

        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30,
0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad,
0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34,
0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07,
0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52,
0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a,
0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45,
0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc,
0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4,
0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46,
0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
```

```
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2,
0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c,
0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8,
0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61,
0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b,
0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41,
0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};


static const uint8_t rsbox[256] = {

        0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf,
0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,

        0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34,
0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

        0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee,
0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,

        0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76,
0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

        0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4,
0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,

        0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e,
0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

        0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7,
0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,

        0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1,
0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,

        0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97,
0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
```

```
        0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2,
0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
        0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f,
0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
        0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a,
0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
        0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1,
0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
        0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d,
0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
        0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8,
0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
        0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1,
0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};


// The round constant word array, Rcon[i], contains the values
given by
// x to th e power (i-1) being powers of x (x is denoted as
{02}) in the field GF(2^8)
static const uint8_t Rcon[11] = {
        0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36};


static uint8_t getSBoxValue(uint8_t num) {
    return sbox[num];
}


static uint8_t getSBoxInvert(uint8_t num) {
    return rsbox[num];
}
```

```
// This function produces Nb(Nr+1) round keys. The round keys are used in each round to decrypt the states.
static void KeyExpansion(void) {
    uint32_t i, k;
    uint8_t tempa[4]; // Used for the column/row operations

    // The first round key is the key itself.
    for (i = 0; i < Nk; ++i) {
        RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
        RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
        RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
        RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
    }

    // All other round keys are found from the previous round keys.
    //i == Nk
    for (; i < Nb * (Nr + 1); ++i) {
        {
            tempa[0] = RoundKey[(i - 1) * 4 + 0];
            tempa[1] = RoundKey[(i - 1) * 4 + 1];
            tempa[2] = RoundKey[(i - 1) * 4 + 2];
            tempa[3] = RoundKey[(i - 1) * 4 + 3];
        }

        if (i % Nk == 0) {
            // This function shifts the 4 bytes in a word to the left once.
            // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]
```

```
            // Function RotWord()
            {
                k = tempa[0];
                tempa[0] = tempa[1];
                tempa[1] = tempa[2];
                tempa[2] = tempa[3];
                tempa[3] = k;
            }

            // SubWord() is a function that takes a four-byte input word and
            // applies the S-box to each of the four bytes to produce an output word.

            // Function Subword()
            {
                tempa[0] = getSBoxValue(tempa[0]);
                tempa[1] = getSBoxValue(tempa[1]);
                tempa[2] = getSBoxValue(tempa[2]);
                tempa[3] = getSBoxValue(tempa[3]);
            }

            tempa[0] = tempa[0] ^ Rcon[i / Nk];
        }
        RoundKey[i * 4 + 0] = RoundKey[(i - Nk) * 4 + 0] ^ tempa[0];
        RoundKey[i * 4 + 1] = RoundKey[(i - Nk) * 4 + 1] ^ tempa[1];
        RoundKey[i * 4 + 2] = RoundKey[(i - Nk) * 4 + 2] ^ tempa[2];
```

```
        RoundKey[i * 4 + 3] = RoundKey[(i - Nk) * 4 + 3] ^
tempa[3];
    }
}

// This function adds the round key to state.
// The round key is added to the state by an XOR function.
static void AddRoundKey(uint8_t round) {
    uint8_t i, j;
    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 4; ++j) {
            (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb
+ j];
        }
    }
}

// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void SubBytes(void) {
    uint8_t i, j;
    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 4; ++j) {
            (*state)[j][i] = getSBoxValue((*state)[j][i]);
        }
    }
}

// The ShiftRows() function shifts the rows in the state to
the left.
```

```
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
static void ShiftRows(void) {
    uint8_t temp;

    // Rotate first row 1 columns to left
    temp = (*state)[0][1];
    (*state)[0][1] = (*state)[1][1];
    (*state)[1][1] = (*state)[2][1];
    (*state)[2][1] = (*state)[3][1];
    (*state)[3][1] = temp;

    // Rotate second row 2 columns to left
    temp = (*state)[0][2];
    (*state)[0][2] = (*state)[2][2];
    (*state)[2][2] = temp;

    temp = (*state)[1][2];
    (*state)[1][2] = (*state)[3][2];
    (*state)[3][2] = temp;

    // Rotate third row 3 columns to left
    temp = (*state)[0][3];
    (*state)[0][3] = (*state)[3][3];
    (*state)[3][3] = (*state)[2][3];
    (*state)[2][3] = (*state)[1][3];
    (*state)[1][3] = temp;
}
```

```
static uint8_t xtime(uint8_t x) {
    return ((x << 1) ^ (((x >> 7) & 1) * 0x1b));
}

// MixColumns function mixes the columns of the state matrix
static void MixColumns(void) {
    uint8_t i;
    uint8_t Tmp, Tm, t;
    for (i = 0; i < 4; ++i) {
        t = (*state)[i][0];
        Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2]
^ (*state)[i][3];
        Tm = (*state)[i][0] ^ (*state)[i][1];
        Tm = xtime(Tm);
        (*state)[i][0] ^= Tm ^ Tmp;
        Tm = (*state)[i][1] ^ (*state)[i][2];
        Tm = xtime(Tm);
        (*state)[i][1] ^= Tm ^ Tmp;
        Tm = (*state)[i][2] ^ (*state)[i][3];
        Tm = xtime(Tm);
        (*state)[i][2] ^= Tm ^ Tmp;
        Tm = (*state)[i][3] ^ t;
        Tm = xtime(Tm);
        (*state)[i][3] ^= Tm ^ Tmp;
    }
}

// Multiply is used to multiply numbers in the field GF(2^8)
#define Multiply(x, y)                                \
```

```
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \


// MixColumns function mixes the columns of the state matrix.
// The method used to multiply may be difficult to understand for the inexperienced.
// Please use the references to gain more information.
static void InvMixColumns(void) {
    int i;
    uint8_t a, b, c, d;
    for (i = 0; i < 4; ++i) {
        a = (*state)[i][0];
        b = (*state)[i][1];
        c = (*state)[i][2];
        d = (*state)[i][3];

        (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d, 0x09);
        (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
        (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
        (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d, 0x0e);
    }
}
```

```
// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void InvSubBytes(void) {
    uint8_t i, j;
    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 4; ++j) {
            (*state)[j][i] = getSBoxInvert((*state)[j][i]);
        }
    }
}

static void InvShiftRows(void) {
    uint8_t temp;

    // Rotate first row 1 columns to right
    temp = (*state)[3][1];
    (*state)[3][1] = (*state)[2][1];
    (*state)[2][1] = (*state)[1][1];
    (*state)[1][1] = (*state)[0][1];
    (*state)[0][1] = temp;

    // Rotate second row 2 columns to right
    temp = (*state)[0][2];
    (*state)[0][2] = (*state)[2][2];
    (*state)[2][2] = temp;

    temp = (*state)[1][2];
```

```
    (*state)[1][2] = (*state)[3][2];
    (*state)[3][2] = temp;

    // Rotate third row 3 columns to right
    temp = (*state)[0][3];
    (*state)[0][3] = (*state)[1][3];
    (*state)[1][3] = (*state)[2][3];
    (*state)[2][3] = (*state)[3][3];
    (*state)[3][3] = temp;
}


// Cipher is the main function that encrypts the PlainText.
static void Cipher(void) {
    uint8_t round = 0;

    // Add the First round key to the state before starting
the rounds.
    AddRoundKey(0);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.
    for (round = 1; round < Nr; ++round) {
        SubBytes();
        ShiftRows();
        MixColumns();
        AddRoundKey(round);
    }
```

```
    // The last round is given below.
    // The MixColumns function is not here in the last round.
    SubBytes();
    ShiftRows();
    AddRoundKey(Nr);
}

static void InvCipher(void) {
    uint8_t round = 0;

    // Add the First round key to the state before starting
the rounds.
    AddRoundKey(Nr);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.
    for (round = (Nr - 1); round > 0; --round) {
        InvShiftRows();
        InvSubBytes();
        AddRoundKey(round);
        InvMixColumns();
    }

    // The last round is given below.
    // The MixColumns function is not here in the last round.
    InvShiftRows();
    InvSubBytes();
```

```
    AddRoundKey(0);
}


static void XorWithIv(char *buf) {
    uint8_t i;
    for (i = 0;
         i < BLOCKLEN; ++i) { //WAS for(i = 0; i < KEYLEN; ++i)
but the block in AES is always 128bit so 16 bytes!
        buf[i] ^= Iv[i];
    }
}

static void clearBuffer(char* buffer) {
    uint8_t i = 0;

    for(i = 0; i < OUTBUFFERSIZE; i++) {
        buffer[i] = 0;
    }
}

uint32_t  AES_CBC_encrypt_buffer(char  *output,  char  *input,
char *key, int length) {
    if(length == -1) {
        length = strlen(input);
    }

    Iv = IV;
    uintptr_t i;
```

```
    uint8_t extra = length % BLOCKLEN; /* Remaining bytes in
the last non-full block */
    uint32_t nblocks = (int)(length / BLOCKLEN);

    clearBuffer(output);

    for (i = 0; i < nblocks; i++) {
        memcpy(output, input, BLOCKLEN);
        XorWithIv(output);
        state = (state_t *) output;
        Cipher();
        Iv = output;
        input += BLOCKLEN;
        output += BLOCKLEN;
        //printf("Step %d - %d", i/16, i);
    }

    if (extra) {
        memcpy(output, input, extra);
        memset((output + extra), 0, (BLOCKLEN - extra));
        XorWithIv(output);
        state = (state_t *) output;
        Cipher();

        nblocks++;
    }

    return BLOCKLEN * nblocks;
}
```

```
uint32_t  AES_CBC_decrypt_buffer(char  *output,  char  *input,
char *key, int length) {
    if(length == -1) {
        length = strlen(input);
    }

    Iv = IV;
    uintptr_t i;
    uint8_t extra = length % BLOCKLEN; /* Remaining bytes in
the last non-full block */
    uint32_t nblocks = (int)(length / BLOCKLEN);

    clearBuffer(output);

    for (i = 0; i < nblocks; i++) {
        memcpy(output, input, BLOCKLEN);
        state = (state_t *) output;
        InvCipher();
        XorWithIv(output);
        Iv = input;
        input += BLOCKLEN;
        output += BLOCKLEN;

    }

    if (extra) {
        memcpy(output, input, extra);
        state = (state_t *) output;
        InvCipher();
```

```
        nblocks++;
    }

    return BLOCKLEN * nblocks;
}

void generate_keys(char *key) {
    Key = key;
    KeyExpansion();
}
```

9.2.2 aes.h

```
#ifndef _AES_H_
#define _AES_H_

#include <stdint.h>

uint32_t  AES_CBC_encrypt_buffer(char  *output,  char  *input,
char *key, int length);

uint32_t  AES_CBC_decrypt_buffer(char  *output,  char  *input,
char *key, int length);

void generate_keys(char *key);

#endif //_AES_H_
```

9.2.3 aes.S

```
defprimitive ">aes",4,eaes,REGULAR
    DPOP a5                     // plaintext_length
    DPOP a4                     // key
    DPOP a3                     // plaintext
    DPOP a2                     // ciphertext
    CCALL AES_CBC_encrypt_buffer
    DPUSH a2                    // ciphertext_length
    NEXT


defprimitive "aes>",4,daes,REGULAR
    DPOP a5                     // ciphertext_length
    DPOP a4                     // key
    DPOP a3                     // ciphertext
    DPOP a2                     // plaintext
    CCALL AES_CBC_decrypt_buffer
    DPUSH a2                    // plaintext_length
    NEXT
```

9.2.4 base64.c

```
#include <string.h>


#include "base64.h"


static const unsigned char pr2six[256] = {
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
```

```
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 62, 64,
64, 64, 63,
        52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 64, 64,
64, 64, 64,
        64, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 64, 64,
64, 64, 64,
        64, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40,
        41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
64, 64, 64
};

int Base64decode_len(const char *bufcoded) {
    int nbytesdecoded;
    register const unsigned char *bufin;
```

```
    register int nprbytes;

    bufin = (const unsigned char *) bufcoded;
    while (pr2six[*(bufin++)] <= 63);

    nprbytes = (bufin - (const unsigned char *) bufcoded) - 1;
    nbytesdecoded = ((nprbytes + 3) / 4) * 3;

    return nbytesdecoded + 1;
}

void Base64decode(char *bufplain, const char *bufcoded) {
    int nbytesdecoded;
    register const unsigned char *bufin;
    register unsigned char *bufout;
    register int nprbytes;

    bufin = (const unsigned char *) bufcoded;
    while (pr2six[*(bufin++)] <= 63);
    nprbytes = (bufin - (const unsigned char *) bufcoded) - 1;
    nbytesdecoded = ((nprbytes + 3) / 4) * 3;

    bufout = (unsigned char *) bufplain;
    bufin = (const unsigned char *) bufcoded;

    while (nprbytes > 4) {
        *(bufout++) =
                (unsigned char) (pr2six[*bufin] << 2 |
pr2six[bufin[1]] >> 4);
```

```
        *(bufout++) =
                (unsigned char) (pr2six[bufin[1]] << 4 |
pr2six[bufin[2]] >> 2);
        *(bufout++) =
                (unsigned char) (pr2six[bufin[2]] << 6 |
pr2six[bufin[3]]);
        bufin += 4;
        nprbytes -= 4;
    }

    /* Note: (nprbytes == 1) would be an error, so just ingore
that case */
    if (nprbytes > 1) {
        *(bufout++) =
                (unsigned char) (pr2six[*bufin] << 2 |
pr2six[bufin[1]] >> 4);
    }
    if (nprbytes > 2) {
        *(bufout++) =
                (unsigned char) (pr2six[bufin[1]] << 4 |
pr2six[bufin[2]] >> 2);
    }
    if (nprbytes > 3) {
        *(bufout++) =
                (unsigned char) (pr2six[bufin[2]] << 6 |
pr2six[bufin[3]]);
    }

    *(bufout++) = '\0';
    nbytesdecoded -= (4 - nprbytes) & 3;
}
```

```
static const char basis_64[] =

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234567
89+/";

int Base64encode_len(int len) {
    return ((len + 2) / 3 * 4) + 1;
}

void Base64encode(char *encoded, const char *string, int len)
{
    int i;
    char *p;

    if(len == -1) {
        len = strlen(string);
    }

    p = encoded;
    for (i = 0; i < len - 2; i += 3) {
        *p++ = basis_64[(string[i] >> 2) & 0x3F];
        *p++ = basis_64[((string[i] & 0x3) << 4) |
                        ((int) (string[i + 1] & 0xF0) >> 4)];
        *p++ = basis_64[((string[i + 1] & 0xF) << 2) |
                        ((int) (string[i + 2] & 0xC0) >> 6)];
        *p++ = basis_64[string[i + 2] & 0x3F];
    }
    if (i < len) {
        *p++ = basis_64[(string[i] >> 2) & 0x3F];
```

```
        if (i == (len - 1)) {
            *p++ = basis_64[((string[i] & 0x3) << 4)];
            *p++ = '=';
        } else {
            *p++ = basis_64[((string[i] & 0x3) << 4) |
                            ((int) (string[i + 1] & 0xF0) >>
4)];
            *p++ = basis_64[((string[i + 1] & 0xF) << 2)];
        }
        *p++ = '=';
    }

    *p++ = '\0';
}
```

9.2.5 base64.h

```
#ifndef _BASE64_H_
#define _BASE64_H_

#ifdef __cplusplus
extern "C" {
#endif

int Base64encode_len(int len);
void Base64encode(char * coded_dst, const char *plain_src, int
len);

int Base64decode_len(const char * coded_src);
void Base64decode(char * plain_dst, const char *coded_src);
```

```
#ifdef __cplusplus
}
#endif

#endif //_BASE64_H_
```

9.2.6 base64.S

```
defprimitive ">base64",7,ebase64,REGULAR
    DPOP a4                     // plaintext_length
    DPOP a3                     // plaintext
    DPOP a2                     // digest
    CCALL Base64encode
    NEXT

defprimitive "base64>",7,dbase64,REGULAR
    DPOP a3                     // digest
    DPOP a2                     // plaintext
    CCALL Base64decode
    NEXT
```

9.2.7 des.c

```
/**************************          HEADER          FILES
**************************/
#include <stdlib.h>
#include <memory.h>
#include <stdint.h>
#include "des.h"
```

```
/*****************************                                    MACROS
*****************************/

// Obtain bit "b" from the left and shift it "c" places from
the right

#define BITNUM(a,b,c) (((a[(b)/8] >> (7 - (b%8))) & 0x01) <<
(c))

#define BITNUMINTR(a,b,c) ((((a) >> (31 - (b))) & 0x00000001)
<< (c))

#define BITNUMINTL(a,b,c) ((((a) << (b)) & 0x80000000) >> (c))

#define OUTBUFFERSIZE 136


// This macro converts a 6 bit block with the S-Box row defined
as the first and last

// bits to a 6 bit block with the row defined by the first two
bits.

#define SBOXBIT(a) (((a) & 0x20) | (((a) & 0x1f) >> 1) | (((a)
& 0x01) << 4))


/*****************************                                    VARIABLES
*****************************/

static const BYTE sbox1[64] = {

        14,  4,  13,  1,   2, 15,  11,  8,   3, 10,   6, 12,
5,  9,   0,  7,

        0, 15,   7,  4,  14,  2,  13,  1,  10,  6,  12, 11,
9,  5,   3,  8,

        4,  1,  14,  8,  13,  6,   2, 11,  15, 12,   9,  7,
3, 10,   5,  0,

        15, 12,   8,  2,   4,  9,   1,  7,   5, 11,   3, 14,
10,  0,   6, 13

};

static const BYTE sbox2[64] = {
```

```
        15,  1,   8, 14,   6, 11,   3,  4,   9,  7,   2, 13,
12,  0,   5, 10,
        3, 13,   4,  7,  15,  2,   8, 14,  12,  0,   1, 10,
6,  9,  11,  5,
        0, 14,   7, 11,  10,  4,  13,  1,   5,  8,  12,  6,
9,  3,   2, 15,
        13,  8,  10,  1,   3, 15,   4,  2,  11,  6,   7, 12,
0,  5,  14,  9
};
static const BYTE sbox3[64] = {
        10,  0,   9, 14,   6,  3,  15,  5,   1, 13,  12,  7,
11,  4,   2,  8,
        13,  7,   0,  9,   3,  4,   6, 10,   2,  8,   5, 14,
12, 11,  15,  1,
        13,  6,   4,  9,   8, 15,   3,  0,  11,  1,   2, 12,
5, 10,  14,  7,
        1, 10,  13,  0,   6,  9,   8,  7,   4, 15,  14,  3,
11,  5,   2, 12
};
static const BYTE sbox4[64] = {
        7, 13,  14,  3,   0,  6,   9, 10,   1,  2,   8,  5,
11, 12,   4, 15,
        13,  8,  11,  5,   6, 15,   0,  3,   4,  7,   2, 12,
1, 10,  14,  9,
        10,  6,   9,  0,  12, 11,   7, 13,  15,  1,   3, 14,
5,  2,   8,  4,
        3, 15,   0,  6,  10,  1,  13,  8,   9,  4,   5, 11,
12,  7,   2, 14
};
static const BYTE sbox5[64] = {
        2, 12,   4,  1,   7, 10,  11,  6,   8,  5,   3, 15,
13,  0,  14,  9,
```

```
        14, 11,   2, 12,   4,  7,  13,  1,   5,  0,  15, 10,
3,  9,   8,  6,
        4,  2,   1, 11,  10, 13,   7,  8,  15,  9,  12,  5,
6,  3,   0, 14,
        11,  8,  12,  7,   1, 14,   2, 13,   6, 15,   0,  9,
10,  4,   5,  3
};
static const BYTE sbox6[64] = {
        12,  1,  10, 15,   9,  2,   6,  8,   0, 13,   3,  4,
14,  7,   5, 11,
        10, 15,   4,  2,   7, 12,   9,  5,   6,  1,  13, 14,
0, 11,   3,  8,
        9, 14,  15,  5,   2,  8,  12,  3,   7,  0,   4, 10,
1, 13,  11,  6,
        4,  3,   2, 12,   9,  5,  15, 10,  11, 14,   1,  7,
6,  0,   8, 13
};
static const BYTE sbox7[64] = {
        4, 11,   2, 14,  15,  0,   8, 13,   3, 12,   9,  7,
5, 10,   6,  1,
        13,  0,  11,  7,   4,  9,   1, 10,  14,  3,   5, 12,
2, 15,   8,  6,
        1,  4,  11, 13,  12,  3,   7, 14,  10, 15,   6,  8,
0,  5,   9,  2,
        6, 11,  13,  8,   1,  4,  10,  7,   9,  5,   0, 15,
14,  2,   3, 12
};
static const BYTE sbox8[64] = {
        13,  2,   8,  4,   6, 15,  11,  1,  10,  9,   3, 14,
5,  0,  12,  7,
        1, 15,  13,  8,  10,  3,   7,  4,  12,  5,   6, 11,
0, 14,   9,  2,
```

```
        7, 11,   4,  1,   9, 12,  14,  2,   0,  6,  10, 13,
15,  3,   5,  8,
        2,  1,  14,  7,   4, 10,   8, 13,  15, 12,   9,  0,
3,  5,   6, 11
};

/**********************          FUNCTION          DEFINITIONS
**********************/
// Initial (Inv)Permutation step
void IP(WORD state[], const BYTE in[])
{
    state[0]   =   BITNUM(in,57,31)   |   BITNUM(in,49,30)   |
BITNUM(in,41,29) | BITNUM(in,33,28) |
               BITNUM(in,25,27)    |    BITNUM(in,17,26)    |
BITNUM(in,9,25) | BITNUM(in,1,24) |
               BITNUM(in,59,23)    |    BITNUM(in,51,22)    |
BITNUM(in,43,21) | BITNUM(in,35,20) |
               BITNUM(in,27,19)    |    BITNUM(in,19,18)    |
BITNUM(in,11,17) | BITNUM(in,3,16) |
               BITNUM(in,61,15)    |    BITNUM(in,53,14)    |
BITNUM(in,45,13) | BITNUM(in,37,12) |
               BITNUM(in,29,11)    |    BITNUM(in,21,10)    |
BITNUM(in,13,9) | BITNUM(in,5,8) |
               BITNUM(in,63,7)     |     BITNUM(in,55,6)     |
BITNUM(in,47,5) | BITNUM(in,39,4) |
               BITNUM(in,31,3)     |     BITNUM(in,23,2)     |
BITNUM(in,15,1) | BITNUM(in,7,0);

    state[1]   =   BITNUM(in,56,31)   |   BITNUM(in,48,30)   |
BITNUM(in,40,29) | BITNUM(in,32,28) |
               BITNUM(in,24,27)    |    BITNUM(in,16,26)    |
BITNUM(in,8,25) | BITNUM(in,0,24) |
```

```
                BITNUM(in,58,23)    |    BITNUM(in,50,22)    |
BITNUM(in,42,21) | BITNUM(in,34,20) |
                BITNUM(in,26,19)    |    BITNUM(in,18,18)    |
BITNUM(in,10,17) | BITNUM(in,2,16) |
                BITNUM(in,60,15)    |    BITNUM(in,52,14)    |
BITNUM(in,44,13) | BITNUM(in,36,12) |
                BITNUM(in,28,11)    |    BITNUM(in,20,10)    |
BITNUM(in,12,9) | BITNUM(in,4,8) |
                BITNUM(in,62,7)     |     BITNUM(in,54,6)    |
BITNUM(in,46,5) | BITNUM(in,38,4) |
                BITNUM(in,30,3)     |     BITNUM(in,22,2)    |
BITNUM(in,14,1) | BITNUM(in,6,0);
}

void InvIP(WORD state[], BYTE in[])
{
    in[0]         =         BITNUMINTR(state[1],7,7)         |
BITNUMINTR(state[0],7,6) | BITNUMINTR(state[1],15,5) |
            BITNUMINTR(state[0],15,4)                        |
BITNUMINTR(state[1],23,3) | BITNUMINTR(state[0],23,2) |
            BITNUMINTR(state[1],31,1)                        |
BITNUMINTR(state[0],31,0);

    in[1]         =         BITNUMINTR(state[1],6,7)         |
BITNUMINTR(state[0],6,6) | BITNUMINTR(state[1],14,5) |
            BITNUMINTR(state[0],14,4)                        |
BITNUMINTR(state[1],22,3) | BITNUMINTR(state[0],22,2) |
            BITNUMINTR(state[1],30,1)                        |
BITNUMINTR(state[0],30,0);

    in[2]         =         BITNUMINTR(state[1],5,7)         |
BITNUMINTR(state[0],5,6) | BITNUMINTR(state[1],13,5) |
```

```
            BITNUMINTR(state[0],13,4)                          |
BITNUMINTR(state[1],21,3) | BITNUMINTR(state[0],21,2) |
            BITNUMINTR(state[1],29,1)                          |
BITNUMINTR(state[0],29,0);


    in[3]         =         BITNUMINTR(state[1],4,7)           |
BITNUMINTR(state[0],4,6) | BITNUMINTR(state[1],12,5) |
            BITNUMINTR(state[0],12,4)                          |
BITNUMINTR(state[1],20,3) | BITNUMINTR(state[0],20,2) |
            BITNUMINTR(state[1],28,1)                          |
BITNUMINTR(state[0],28,0);


    in[4]         =         BITNUMINTR(state[1],3,7)           |
BITNUMINTR(state[0],3,6) | BITNUMINTR(state[1],11,5) |
            BITNUMINTR(state[0],11,4)                          |
BITNUMINTR(state[1],19,3) | BITNUMINTR(state[0],19,2) |
            BITNUMINTR(state[1],27,1)                          |
BITNUMINTR(state[0],27,0);


    in[5]         =         BITNUMINTR(state[1],2,7)           |
BITNUMINTR(state[0],2,6) | BITNUMINTR(state[1],10,5) |
            BITNUMINTR(state[0],10,4)                          |
BITNUMINTR(state[1],18,3) | BITNUMINTR(state[0],18,2) |
            BITNUMINTR(state[1],26,1)                          |
BITNUMINTR(state[0],26,0);


    in[6]         =         BITNUMINTR(state[1],1,7)           |
BITNUMINTR(state[0],1,6) | BITNUMINTR(state[1],9,5) |
            BITNUMINTR(state[0],9,4)                           |
BITNUMINTR(state[1],17,3) | BITNUMINTR(state[0],17,2) |
            BITNUMINTR(state[1],25,1)                          |
BITNUMINTR(state[0],25,0);
```

```
    in[7]         =         BITNUMINTR(state[1],0,7)         |
BITNUMINTR(state[0],0,6) | BITNUMINTR(state[1],8,5) |
            BITNUMINTR(state[0],8,4)                          |
BITNUMINTR(state[1],16,3) | BITNUMINTR(state[0],16,2) |
            BITNUMINTR(state[1],24,1)                         |
BITNUMINTR(state[0],24,0);
}

WORD f(WORD state, const BYTE key[])
{
    BYTE lrgstate[6]; //,i;
    WORD t1,t2;

    // Expantion Permutation
    t1 = BITNUMINTL(state,31,0) | ((state & 0xf0000000) >> 1)
| BITNUMINTL(state,4,5) |
         BITNUMINTL(state,3,6) | ((state & 0x0f000000) >> 3) |
BITNUMINTL(state,8,11) |
         BITNUMINTL(state,7,12) | ((state & 0x00f00000) >> 5)
| BITNUMINTL(state,12,17) |
         BITNUMINTL(state,11,18) | ((state & 0x000f0000) >> 7)
| BITNUMINTL(state,16,23);

    t2 = BITNUMINTL(state,15,0) | ((state & 0x0000f000) << 15)
| BITNUMINTL(state,20,5) |
         BITNUMINTL(state,19,6) | ((state & 0x00000f00) << 13)
| BITNUMINTL(state,24,11) |
         BITNUMINTL(state,23,12) | ((state & 0x000000f0) <<
11) | BITNUMINTL(state,28,17) |
         BITNUMINTL(state,27,18) | ((state & 0x0000000f) << 9)
| BITNUMINTL(state,0,23);
```

```
    lrgstate[0] = (t1 >> 24) & 0x000000ff;
    lrgstate[1] = (t1 >> 16) & 0x000000ff;
    lrgstate[2] = (t1 >> 8) & 0x000000ff;
    lrgstate[3] = (t2 >> 24) & 0x000000ff;
    lrgstate[4] = (t2 >> 16) & 0x000000ff;
    lrgstate[5] = (t2 >> 8) & 0x000000ff;

    // Key XOR
    lrgstate[0] ^= key[0];
    lrgstate[1] ^= key[1];
    lrgstate[2] ^= key[2];
    lrgstate[3] ^= key[3];
    lrgstate[4] ^= key[4];
    lrgstate[5] ^= key[5];

    // S-Box Permutation
    state = (sbox1[SBOXBIT(lrgstate[0] >> 2)] << 28) |
            (sbox2[SBOXBIT(((lrgstate[0] & 0x03) << 4) |
(lrgstate[1] >> 4))] << 24) |
            (sbox3[SBOXBIT(((lrgstate[1] & 0x0f) << 2) |
(lrgstate[2] >> 6))] << 20) |
            (sbox4[SBOXBIT(lrgstate[2] & 0x3f)] << 16) |
            (sbox5[SBOXBIT(lrgstate[3] >> 2)] << 12) |
            (sbox6[SBOXBIT(((lrgstate[3] & 0x03) << 4) |
(lrgstate[4] >> 4))] << 8) |
            (sbox7[SBOXBIT(((lrgstate[4] & 0x0f) << 2) |
(lrgstate[5] >> 6))] << 4) |
            sbox8[SBOXBIT(lrgstate[5] & 0x3f)];

    // P-Box Permutation
```

```
    state = BITNUMINTL(state,15,0) | BITNUMINTL(state,6,1) |
BITNUMINTL(state,19,2) |
            BITNUMINTL(state,20,3) | BITNUMINTL(state,28,4) |
BITNUMINTL(state,11,5) |
            BITNUMINTL(state,27,6) | BITNUMINTL(state,16,7) |
BITNUMINTL(state,0,8) |
            BITNUMINTL(state,14,9) | BITNUMINTL(state,22,10) |
BITNUMINTL(state,25,11) |
            BITNUMINTL(state,4,12) | BITNUMINTL(state,17,13) |
BITNUMINTL(state,30,14) |
            BITNUMINTL(state,9,15) | BITNUMINTL(state,1,16) |
BITNUMINTL(state,7,17) |
            BITNUMINTL(state,23,18) | BITNUMINTL(state,13,19)
| BITNUMINTL(state,31,20) |
            BITNUMINTL(state,26,21) | BITNUMINTL(state,2,22) |
BITNUMINTL(state,8,23) |
            BITNUMINTL(state,18,24) | BITNUMINTL(state,12,25)
| BITNUMINTL(state,29,26) |
            BITNUMINTL(state,5,27) | BITNUMINTL(state,21,28) |
BITNUMINTL(state,10,29) |
            BITNUMINTL(state,3,30) | BITNUMINTL(state,24,31);


    // Return the final state value
    return(state);
}


void  des_key_setup(const  BYTE  key[],  BYTE  schedule[][6],
DES_MODE mode)
{
    WORD i, j, to_gen, C, D;
    const          WORD          key_rnd_shift[16]          =
{1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
```

```
    const           WORD           key_perm_c[28]           =
{56,48,40,32,24,16,8,0,57,49,41,33,25,17,

9,1,58,50,42,34,26,18,10,2,59,51,43,35};
    const           WORD           key_perm_d[28]           =
{62,54,46,38,30,22,14,6,61,53,45,37,29,21,

13,5,60,52,44,36,28,20,12,4,27,19,11,3};
    const         WORD         key_compression[48]         =
{13,16,10,23,0,4,2,27,14,5,20,9,

22,18,11,3,25,7,15,6,26,19,12,1,

40,51,30,36,46,54,29,39,50,44,32,47,

43,48,38,55,33,52,45,41,49,35,28,31};


    // Permutated Choice #1 (copy the key in, ignoring parity
bits).
    for (i = 0, j = 31, C = 0; i < 28; ++i, --j)
        C |= BITNUM(key,key_perm_c[i],j);
    for (i = 0, j = 31, D = 0; i < 28; ++i, --j)
        D |= BITNUM(key,key_perm_d[i],j);


    // Generate the 16 subkeys.
    for (i = 0; i < 16; ++i) {
        C   =   ((C   <<   key_rnd_shift[i])   |   (C   >>   (28-
key_rnd_shift[i]))) & 0xfffffff0;
        D   =   ((D   <<   key_rnd_shift[i])   |   (D   >>   (28-
key_rnd_shift[i]))) & 0xfffffff0;
```

```
        // Decryption subkeys are reverse order of encryption
subkeys so
        // generate them in reverse if the key schedule is for
decryption useage.
        if (mode == DES_DECRYPT)
            to_gen = 15 - i;
        else /*(if mode == DES_ENCRYPT)*/
            to_gen = i;
        // Initialize the array
        for (j = 0; j < 6; ++j)
            schedule[to_gen][j] = 0;
        for (j = 0; j < 24; ++j)
            schedule[to_gen][j/8]                              |=
BITNUMINTR(C,key_compression[j],7 - (j%8));
        for ( ; j < 48; ++j)
            schedule[to_gen][j/8]                              |=
BITNUMINTR(D,key_compression[j] - 28,7 - (j%8));
    }
}

void  des_crypt(const  BYTE  in[],  BYTE  out[],  const  BYTE
key[][6])
{
    WORD state[2],idx,t;

    IP(state,in);

    for (idx=0; idx < 15; ++idx) {
        t = state[1];
        state[1] = f(state[1],key[idx]) ^ state[0];
```

```
        state[0] = t;
    }
    // Perform the final loop manually as it doesn't switch
sides
    state[0] = f(state[1],key[15]) ^ state[0];

    InvIP(state,out);
}



void clearBuffer(uint8_t* buffer) {
    uint8_t i = 0;

    for(i = 0; i < OUTBUFFERSIZE; i++) {
        buffer[i] = 0;
    }
}

uint32_t des(uint8_t* input, uint8_t* output, uint8_t* key,
int16_t length, DES_MODE action) {
    if(length == -1) {
        length = strlen(input);
    }

    clearBuffer(output);

    uint32_t IV[8] = {0, 0, 0, 0, 0, 0, 0, 0};

    uint8_t block[8];
    uint8_t block_buf[8];
```

```
uint8_t blocks_length   = (length + 8 - 1) / 8;
uint8_t padding_length  = length % 8;

if(padding_length > 0) {
    padding_length = 8 - padding_length;
}

uint8_t schedule[16][6];

des_key_setup(key, schedule, action);

int8_t i;
int8_t j;

for(i = 0; i < blocks_length; i++) {
    if(i == blocks_length - 1 && padding_length > 0) {
        for(j = 0; j < 8 - padding_length; j++) {
            block[j] = input[i * 8 + j];
        }

        for (j = 8 - padding_length; j < 8; j++) {
            block[j] = 0;
        }
    } else {
        for(j = 0; j < 8; j++) {
            block[j] = input[i * 8 + j];
        }
    }
```

```
if(action == DES_ENCRYPT) {
    block[0] = block[0] ^ IV[0];
    block[1] = block[1] ^ IV[1];
    block[2] = block[2] ^ IV[2];
    block[3] = block[3] ^ IV[3];
    block[4] = block[4] ^ IV[4];
    block[5] = block[5] ^ IV[5];
    block[6] = block[6] ^ IV[6];
    block[7] = block[7] ^ IV[7];

    des_crypt(block, block_buf, schedule);

    IV[0] = block_buf[0];
    IV[1] = block_buf[1];
    IV[2] = block_buf[2];
    IV[3] = block_buf[3];
    IV[4] = block_buf[4];
    IV[5] = block_buf[5];
    IV[6] = block_buf[6];
    IV[7] = block_buf[7];
} else {
    des_crypt(block, block_buf, schedule);

    block_buf[0] = block_buf[0] ^ IV[0];
    block_buf[1] = block_buf[1] ^ IV[1];
    block_buf[2] = block_buf[2] ^ IV[2];
    block_buf[3] = block_buf[3] ^ IV[3];
    block_buf[4] = block_buf[4] ^ IV[4];
```

```
            block_buf[5] = block_buf[5] ^ IV[5];
            block_buf[6] = block_buf[6] ^ IV[6];
            block_buf[7] = block_buf[7] ^ IV[7];

            IV[0] = block[0];
            IV[1] = block[1];
            IV[2] = block[2];
            IV[3] = block[3];
            IV[4] = block[4];
            IV[5] = block[5];
            IV[6] = block[6];
            IV[7] = block[7];
        }

        for(j = 0; j < 8; j++) {
            output[i * 8 + j] = block_buf[j];
        }
    }

    return blocks_length * 8;
}

uint32_t des_encrypt(uint8_t *output, uint8_t *input, uint8_t*
key, uint16_t length) {
    return des(input, output, key, length, DES_ENCRYPT);
}

uint32_t des_decrypt(uint8_t *output, uint8_t *input, uint8_t*
key, uint16_t length) {
    return des(input, output, key, length, DES_DECRYPT);
```

```
}
```

9.2.8 des.h

```
#ifndef DES_H

#define DES_H


/**************************          HEADER          FILES
**************************/

#include <stddef.h>


/*****************************                       MACROS
*****************************/

#define DES_BLOCK_SIZE 8                // DES operates on 8
bytes at a time


/**************************          DATA          TYPES
**************************/

typedef unsigned char BYTE;             // 8-bit byte

typedef unsigned int  WORD;             // 32-bit word, change
to "long" for 16-bit machines


typedef enum {

    DES_ENCRYPT,

    DES_DECRYPT

} DES_MODE;


/***********************        FUNCTION        DECLARATIONS
**********************/

void  des_key_setup(const  BYTE  key[],  BYTE  schedule[][6],
DES_MODE mode);
```

```
void des_crypt(const BYTE in[], BYTE out[], const BYTE key[][6]);


void clearBuffer(uint8_t* buffer);


uint32_t des(uint8_t* input, uint8_t* output, uint8_t* key, int16_t length, DES_MODE action);


uint32_t des_encrypt(uint8_t *output, uint8_t *input, uint8_t* key, uint16_t length);
uint32_t des_decrypt(uint8_t *output, uint8_t *input, uint8_t* key, uint16_t length);


#endif   // DES_H
```

9.2.9 des.S

```
defprimitive ">des",4,edes,REGULAR
    DPOP a5                     // plaintext_length
    DPOP a4                     // key
    DPOP a3                     // plaintext
    DPOP a2                     // ciphertext
    CCALL des_encrypt
    DPUSH a2                    // ciphertext_length
    NEXT
```

```
defprimitive "des>",4,ddes,REGULAR
    DPOP a5                     // ciphertext_length
    DPOP a4                     // key
    DPOP a3                     // ciphertext
    DPOP a2                     // plaintext
    CCALL des_decrypt
    DPUSH a2                    // plaintext_length
    NEXT
```

9.2.10 tcp-client.forth

```
0 init-variable: socket
0 task: netfetch-task

128 buffer: line

: tcp-receive ( -- )
  activate
    begin
      socket @ 128 line netcon-readln -1 <>
      line "quit" =str invert and
    while
      line strlen if line eval then
    repeat
    drop

    socket @ netcon-dispose
    0 socket !
  deactivate
;
```

```
: tcp-send ( str -- )
  socket @ if
    socket @ swap netcon-writeln
  then
;

: tcp-disconnect ( -- )
  "quit" tcp-send
;

: tcp-connect ( port ip -- )
  multi
  0 socket !
  TCP netcon-connect socket !
  netfetch-task tcp-receive
;

create: tcp 4 cells allot
' tcp-connect tcp !
' tcp-disconnect tcp 1 cells + !
' tcp-send tcp 2 cells + !

/end
```

9.2.11 mqtt-client.forth

```
0 init-variable: mqtt-socket
0 task: mqttread-task
0 task: ping-task
```

```
: broker 1883 "192.168.0.19" ;

: mqtt-connect ( port ip -- )
  buffer
    $ 10 c!, $ 12 c!, $ 00 c!, $ 06 c!,
    $ 4D c!, $ 51 c!, $ 49 c!, $ 73 c!,
    $ 64 c!, $ 70 c!, $ 03 c!, $ 02 c!,
    $ 00 c!, $ 3C c!, $ 00 c!, $ 04 c!,
    $ 65 c!, $ 73 c!, $ 70 c!, $ 31 c!,
  drop

  TCP ['] netcon-connect catch 0 = if
    dup buffer 20 netcon-write-buf
    dup 4 buffer netcon-read-fix drop

    mqtt-socket !
  else
    3drop
    "** Connection error **" type cr cr
  then
;

: mqtt-disconnect ( -- )
  buffer
    $ E0 c!, $ 00 c!,
  drop

  mqtt-socket @ buffer 2 netcon-write-buf
```

```
  2000 ms
  mqtt-socket @ netcon-dispose

  0 mqtt-socket !
;

: mqtt-ping ( -- )
  buffer
    $ C0 c!, $ 00 c!,
  drop

  mqtt-socket @ buffer 2 netcon-write-buf
;

: mqtt-send ( msg topic -- )
  buffer
    $ 30 c!, $ 02 c!, $ 00 c!, $ 00 c!,

    swap dup -rot buf!
    swap strlen buffer 3 + c@ + buffer 3 + c!

    swap dup -rot buf!
    swap strlen buffer 1 + c@ + buffer 3 + c@ + buffer 1 + c!
  buffer -

  mqtt-socket @ buffer rot netcon-write-buf

  \ TODO ricezione PUBACK
;
```

```
: mqttread ( -- length | -1 se disconesso )
  mqtt-socket @ 1 buffer netcon-read-fix dup -1 <> if \ code
    drop
    mqtt-socket @ 1 buffer netcon-read-fix dup -1 <> if \
length
      drop
      buffer c@ dup 0 > if
        dup 0 do
          mqtt-socket @ 1 buffer i + netcon-read-fix -1 = if
            -1
            unloop exit
          then
        loop
      then
    then
  then

  \  dup 0 > if 0 do buffer i + c@ . cr loop else drop then
;

: mqttread-worker ( -- )
  activate
    begin
      ['] mqttread catch drop dup -1 <>
    while
      dup 0 > swap dup SIZE < rot and if
        0 do buffer i + c@ emit cr loop
      then
```

```
      cr \ TODO eval
    repeat
    drop
  deactivate

  0 mqtt-socket !
;

: mqtt-ping-worker ( -- )
  activate
    begin
      mqtt-socket 0 <>
    while
      30000 ms
      mqtt-ping
    repeat
  deactivate
;

: mqtt-receive ( topic -- )
  buffer
    $ 82 c!, $ 04 c!,
    $ 00 c!, $ 0A c!,
    $ 00 c!, $ 00 c!,
    swap dup -rot buf!
    $ 00 c!,

    swap strlen buffer 5 + c@ + buffer 5 + c!
    buffer 1 + c@ 1 + buffer 5 + c@ + buffer 1 + c!
```

```
  buffer -

  mqtt-socket @ buffer rot netcon-write-buf

  5 0 do
    mqtt-socket @ 1 buffer netcon-read-fix drop
  loop

  multi
  mqttread-task mqttread-worker
\  ping-task mqtt-ping-worker
;

create: mqtt 4 cells allot
' mqtt-connect mqtt !
' mqtt-disconnect mqtt 1 cells + !
' mqtt-send mqtt 2 cells cells + !
' mqtt-receive mqtt 3 cells + !

/end
```

9.2.12 utils.forth

```
: $ immediate
    word hex>int'
    interpret? invert if postpone: literal then
;

\ CONF indica il blocco in flash per leggere / scrivere la
configurazione per IoT
```

```
256 constant: CONF

257 constant: TMP

SIZE buffer: buffer


\ Ottiene l'indirizzo in byte del blocco (block)

\ ( SIZE = 4096, definito in flash.forth)

: addr ( block -- addr )

  SIZE *

;


: c!, ( buffer c -- bufferinc )

  swap dup 1 + -rot c!

;


\ Copia i caratteri presenti in data nel buffer

\ Lascia in stack:

\ - bufferinc: l'indirizzo del buffer incrementato di offset
posizioni

: buf! ( buffer data -- bufferinc)

  dup strlen dup >r


  0 do

    dup i + c@

    rot dup -rot

    i + c!
```

```
    swap
  loop

  drop r> +
;

: clr ( buffer -- )
    dup strlen 0 do
        dup 0 swap i + c!
    loop

    drop
;

\ Cancella un blocco (block) di flash
: wp ( block -- )
  erase-flash drop
;

\ Scrive la stringa (data) in un blocco (block)
: wr ( data block -- )
  dup wp addr
  swap dup strlen
  swap rot
  write-flash drop
```

```
;


\ Legge size caratteri da address in flash e li copia in
buffer

: rd ( size buffer address -- )

  read-flash drop

;


\ Legge da block il primo byte per verificare se block
contiene dati

\ Per il momento, si assume che il valore 255 nel primo byte
di block indichi

\ che la configurazione non sia caricata.

\ Lascia in stack:

\ - block: il blocco su cui si è fatta l'interrogazione

\ - bool: 0 o -1

: ready? ( block -- block bool )

  dup addr

  1 buffer rot rd

  buffer c@ 255 <>

;


\ Effettua la load di block se bool è true

: ?load ( block bool -- )

  if load else drop then

;
```

```
\ Aggiunge
\ \r\n
\ /end
\ \r\n\r\n
\ alla fine di buffer
: >module ( buffer -- )
  dup strlen +
  13 c!, 10 c!,
  47 c!, 101 c!, 110 c!, 100 c!,
  13 c!, 10 c!, 13 c!, 10 c!,
  drop
;

: conf! ( buffer -- )
  dup >module
  CONF wr
;

: [execute]
  cells + @ execute
;

: connect ( addr -- )
  0 [execute]
```

```
;

: disconnect ( addr -- )
  1 [execute]
;

: send ( addr -- )
  2 [execute]
;

: receive ( addr -- )
  3 [execute]
;

/end
```

9.2.13 main.forth

```
FLASH load
GPIO load

CONF ready? ?load

TCPREPL load
TCPCLI load
MQTTCLI load

repl-start
```

9.3 Test

9.3.1 Codice generato per la configurazione

```
: CONSTANT constant: ;
: ROT rot ;
: DROP drop ;
: DUP dup ;
: MS ms ;
: EMIT emit ;
: VARIABLE variable: ;
: HIGH GPIO_HIGH gpio-write ;
: LOW GPIO_LOW gpio-write ;
: SET_MODE gpio-mode ;
: READ adc-read ;
: ON HIGH ;
: OFF LOW ;
12 CONSTANT RLED
RLED GPIO_OUT SET_MODE
14 CONSTANT GLED
GLED GPIO_OUT SET_MODE
: LUMSENS nop ;
store
```

9.3.2 Script Python per il calcolare il tempo di configurazione (client.py)

```
import socket
import time
import datetime


TCP_IP = '192.168.4.2'
TCP_PORT = 1983
BUFFER_SIZE = 128


lines = []
```

```
fp = open("test_conf.forth")
for line in fp:
    lines.append(line.strip())
fp.close()

lines.append("store")

count = 0
for line in lines:
    count += len(line) + 2

out = [
    "N",
    "C",
    "L",
    "T"
]

print "\t".join(out)

for x in range(0, 100):
    t1 = datetime.datetime.now()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((TCP_IP, TCP_PORT))

    for line in lines:
```

```
        s.send(line + "\r\n")

    s.send('quit\r\n')
    data = s.recv(BUFFER_SIZE)
    s.close()

    t2 = datetime.datetime.now()

    delta_t = t2 - t1

    sec = float(delta_t.seconds) + float(delta_t.microseconds) /
float(1000000)

    out = [
        str(x + 1),
        str(count),
        str(len(lines)),
        str(sec).replace(".", ",")
    ]

    print "\t".join(out)
```

9.3.3 TCP Client - Codice simbolico per definire il comportamento di esp2

```
: check
  dup . 200 > if
    YLED ON
  else
    YLED OFF
  then
```

```
;
0 task: test-task
1 mailbox: test-mailbox
0 init-variable: test-running
: test
  activate
  begin
    test-running
  while
    "LUMSENS READ . tcp-response return\r\n" tcp send
    tcpcli-mailbox mailbox-receive check
    pause
  repeat
  deactivate
;
: test-start
  1983 "192.168.4.2" tcp connect
  -1 test-running !
  test-task test
;
: test-stop
  tcp disconnect
  0 test-running !
;
```

Riferimenti Bibliografici:

[1] K. Ashton, «‘That ‘Internet of Things’ thing,’’ RFID J., Jul. 2009.,» [Online]. Available: http://www.rfidjournal.com/article/view/4986.

[2] J. KILJANDER, A. D'ELIA, F. MORANDI, P. HYTTINEN, J. TAKALO-MATTILA, A. YLISAUKKO-OJA, J.-P. SOININEN e T. S. CINOTTI, «Semantic Interoperability Architecture for Pervasive Computing and Internet of Things,» 2014.

[3] M. Kovatsch, S. Mayer e B. Ostermaier, «Moving application logic from the firmware to the cloud: Towards the thin server architecture for the Internet of Things. In Proceedings of the 2012 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing,» 2012.

[4] C. Mille e C. Poellabauer, «Reliable and efficient reprogramming in sensor networks. ACM Trans- actions on Sensor Networks 7,» 2010.

[5] M. H. A. Waqaas Munawar, O. Landsiedel e K. Wehrle., «Dynamic tinyOS: Modular and transparent incremental code-updates for sensor networks,» 2010.

[6] S. Gaglio, G. Lo Re, G. Martorella e D. Peri, «DC4CD: A Platform for Distributed Computing on Constrained Devices,» 2017.

[7] T. Salman, «Networking Protocols and Standards for Internet of Things».

[8] C. Gomez, J. Oller e J. Paradells, «Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology,» 2012.

[9] M. Park, «IEEE 802.11ah: sub-1-GHz license-exempt operation for the internet of things,» 2015.

[10] Internet Engineering Task Force (IETF), RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, 2012.

[11] A. Aijaz e A. H. Aghvami, «Cognitive Machine-to-Machine Communications for Internet-of-Things: A Protocol Stack Perspective,» 2015.

[12] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego e J. Alonso-Zarate, «A Survey on Application Layer Protocols for the Internet of Things,» 2015.

[13] Object Management Group, «ABOUT THE DATA DISTRIBUTION SERVICE SPECIFICATION VERSION 1.4,» 2015. [Online]. Available: https://www.omg.org/spec/DDS/1.4.

[14] Espressif System, «ESP8266EX Datasheet,» 2018.

[15] Tensilica Inc, «Xtensa® Instruction Set Architecture (ISA),» 2010.

[16] Tensilica Inc, «Diamond Standard Processor Cores».

[17] A. Magyar, «Punyforth,» [Online]. Available: https://github.com/zeroflag/punyforth.

[18] Espressif, «esptool.py,» [Online]. Available: https://github.com/espressif/esptool.

[19] D. Locke, «MQ telemetry transport (MQTT) v3. 1 protocol specification,» 2010. [Online]. Available: https://www.ibm.com/developerworks/webservices/library/ws-mqtt/index.html.

[20] IBM, «MQTT V3.1 Protocol Specification,» [Online]. Available: http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#utf-8.

[21] Espressif System, «ESP8266_RTOS_SDK v2.0.0».

Indice delle tabelle

Tabella 1 - Protocolli Livello Applicazione 5
Tabella 2 - Specifiche ESP8266 16
Tabella 3 - Pin ESP8266-12E 18
Tabella 4 - Mappatura Pin NodeMCU Amica e ESP8266-12E 48
Tabella 5 - Porte Seriali per Sistemi Operativi 50
Tabella 6 - Parametri flash.py 51
Tabella 7 - Modalità di accesso alla flash SPI per ESP8266 e ESP32 53
Tabella 8 - Header fisso del protocollo MQTT 75
Tabella 9 - Comandi utilizzati dal MQTT 75
Tabella 10 - Occupazione memoria flash SPI 81
Tabella 11 - Memoria RAM occupata e disponibile 82
Tabella 12 - Dimensioni delle immagini binarie 84
Tabella 13 - Dimensione moduli scritti in Codice Simbolico 84
Tabella 14 - Valore minimo, medio e massimo dei tempi di trasmissione per byte trasmessi 87
Tabella 15 - Risultato confronto ESP-RTOS-MQTT e mqtt-client.forth 93

Indice delle figure

Figura 1 - Stack dei Protocolli IoT 3
Figura 2 - Scambio di informazioni tra le componenti del Sistema 6
Figura 3 - Struttura ad albero del mondo fisico mediante il quale è possibile identificare oggetti, locazioni e stati 7
Figura 4 - Hardware utilizzato nel progetto in esame 9
Figura 5 - Interazione tra le componenti software del progetto in esame 11
Figura 6 - Schema dell'Ambiente Simbolico utilizzato nel progetto in esame 12
Figura 7 - ESP8266-12E 14
Figura 8 - ESP8266 14
Figura 9 - NodeMCU Amica DevBoard - ESP8266-12E 14
Figura 10 - Memorizzazione e lettura di codice simbolico nella flash SPI 43
Figura 11 - Caricamento del contenuto del blocco di flash SPI dopo il riavvio 44
Figura 12 - NodeMCU Amica pin header 49
Figura 13 - TCP REPL - Funzionamento della parola command-loop 61
Figura 14 - Interazione Sistema di Regole e Ambiente Simbolico 64
Figura 15 - Cifratura e Codifica di una stringa di codice simbolico utilizzando DES-CBC e Base64 65
Figura 16 - Interazione Client TCP e Server TCP REPL 72
Figura 17 - Scambio di messaggi tramite il protocollo MQTT 74
Figura 18 - Dimensioni immagini binarie e moduli 83
Figura 19 - Collegamenti LED e sensore su myesp 85
Figura 20 - Invio, ricezione e memorizzazione della configurazione 86
Figura 21 - Esecuzione azione al riavvio 86
Figura 22 - Grafico dei tempi di trasmissione di codice simbolico 88
Figura 23 - Collegamenti di esp1 e esp2 89
Figura 24 - Comunicazione tra nodi tramite TCP REPL e TCP Client 90
Figura 25 - Test di invio e ricezione di codice simbolico tramite protocollo MQTT 92
Figura 26 - Grafico confronto ESP-RTOS-MQTT e mqtt-client.forth 93