ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση ΗΚΓ σε ενσωματώμενες εφαρμογές IoT με Λειτουργικά Πραγματικού Χρόνου (RTOS)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΑΝΤΩΝΙΟΥ Ν. ΚΕΚΕΜΠΑΝΟΥ

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση ΗΚΓ σε ενσωματώμενες εφαρμογές IoT με Λειτουργικά Πραγματικού Χρόνου (RTOS)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΑΝΤΩΝΙΟΥ Ν. ΚΕΚΕΜΠΑΝΟΥ

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Φεβρουαρίου 2018.

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Πεκμεστζή	Κώστας Σιόζος
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Α.Π.Θ.

Αθήνα, Φεβρουάριος 2018

.....

ΑΝΤΩΝΙΟΣ Ν. ΚΕΚΕΜΠΑΝΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

© 2018 – All rights reserved

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη της ανάγκης χρήσης του λειτουργικών συστημάτων πραγματικού χρόνου στις εφαρμογές του διαδικτύου των πραγμάτων (Internet of Things), η μελέτη του λειτουργικού συστήματος πραγματικού χρόνου FreeRTOS, η πειραματική εφαρμογή του στην αναπτυξιακή πλακέτα Digilent Zybo καθώς και η ανάπτυξη εφαρμογής ανάλυσης καρδιογραφήματος ΗΚΓ (ECG) στην αναπτυξιακή πλακέτα Particle Photon.

Αρχικά καταγράφονται οι συνθήκες στο περιβάλλον του διαδικτύου των πραγμάτων (Internet of Things, IoT) με ιδιαίτερη αναφορά στις προκλήσεις που ενέχει ο σχεδιασμός ενός IoT προϊόντος. Έπειτα, αναλύεται το πως η χρήση ενός κατάλληλου λειτουργικού συστήματος πραγματικού χρόνο (Real time Operating System, RTOS) μπορεί να δώσει λύση στην αντιμετώπιση αυτών των προκλήσεων και να περιορίσει σημαντικά το χρόνο ανάπτυξης. Τέλος, παραθέτονται και σχετικά παραδείγματα από την βιβλιογραφία στα οποία έχει μελετηθεί η χρήση RTOS στο IoT.

Στην συνέχεια, γίνεται μια συνοπτική παρουσίαση του FreeRTOS που εκθέτει το πεδίο χρήσης του, τα κυριότερα χαρακτηριστικά του και την θέση που κατέχει στην αγορά σήμερα. Έπειτα αναφέρονται οι βασικότερες έννοιες του με παραδείγματα της προγραμματιστικής διεπαφής που παρέχει. Στη συνέχεια ακολουθεί μια συνοπτική σύγκριση μεταξύ FreeRTOS και Linux που αφορά τα κυριότερα χαρακτηριστικά τους ώστε να γίνει αντιληπτό το πεδίο χρήσης του καθενός λειτουργικού.

Μετά την θεωρητική μελέτη του FreeRTOS αναπτύχθηκε μια εφαρμογή επίδειξης σην αναπτυξιακή πλακέτα Digilent Zybo για την παρουσίαση της λειτουργίας του FreeRTOS. Το Zybo χρησιμοποιεί το Xilinx Zynq-7000 All Programmable SoC και γίνεται περιγραφή τόσο της εγκατάστασης του FreeRTOS όσο και της προετοιμασίας του υλικού που αφορά τον προγραμματισμό του FPGA για την σύνδεση των περιφερειακών.

Τέλος παρουσιάζεται η ανάπτυξη της εφαρμογής καταγραφής κι ανάλυσης καρδιογραφήματος ΗΚΓ (ECG) στο Particle Photon. Η εφαρμογή αυτή βασίζεται σε προηγούμενη εργασία [5] του εργαστηρίου και το σημείο αφετηρίας για την ανάπτυξη στο Particle Photon είναι ο έτοιμος κώδικας C ο οποίος όμως προορίζεται για εκτέλεση σε περιβάλλον Linux. Έτσι παρουσιάζονται τα στάδια της μετατροπής του για εκτέλεση στο Photon και στη συνέχεια η προσθήκη της δυνατότητας καταγραφής πραγματικού σήματος καρδιογραφήματος ECG σε πραγματικό χρόνο ως επίσης και η αποστολή του σήματος μαζί με τα αποτελέσματα της ανάλυσης σε οποιοδήποτε σταθμό μέσω διαδικτύου.

Λέξεις Κλειδιά: Διαδίκτυο των Πραγμάτων, IoT, Λειτουργικό πραγματικού χρόνου, FreeRTOS, Linux, σύγκριση, Xilinx, Zynq-7000 SOC, Digilent Zybo, Particle Photon, ανάλυση, ηλεκτροκαρδιογραφήμα, ΗΚΓ, ECG

Abstract

The scope of this thesis is the study of the key challenges in development of applications for the IoT environment and how the use of a Real Time Operating System can help to overcome these difficulties, the study of FreeRTOS accompanied with a demo application on Digilent Zybo and the porting of an ECG analysis application to Particle Photon development board.

After a general description of the current IoT environment we focus on the main challenges in making a competitive IoT product. By taking one by one these key points we demonstrate how the use of a suitable RTOS can easily fulfill these design goals and minimize the development time.

Among RTOSs FreeRTOS is the industry leading one which justifies its selection for our study. We present its main features and its current market position. Then we describe the main programming concepts in FreeRTOS with API examples where needed. A comparison between FreeRTOS and Linux is also included in order to highlight the main differences between them and give guidelines on the selection of the appropriate OS for specific applications.

The development of a FreeRTOS demo application on Digilent Zybo development board was also conducted. We describe the porting of FreeRTOS to Xilinx Zynq-7000 All Programmable SoC which is incorporated in Zybo board as well as the FPGA programming which was necessary in order to connect the board peripherals to the ARM core inside Zynq.

Finally, the development of the ECG analysis on Particle Photon is presented. The starting point was the C code of the application developed by a previous thesis [5]. This code was intended to run on a Linux environment so there was the requirement for major changes to make porting to Photon viable. Real time signal sampling and networking features were later added which allow the application to send sampled signal windows and corresponding analysis results to any station via Internet.

Keywords: Internet of Things, Real Time Operating System, FreeRTOS, Linux, comparison, Xilinx, Zynq-7000 SoC, Digilent, Zybo, Particle, Photon, ECG analysis

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θέρμα τον κ. Δημήτριο Σούντρη, Αναπλ.Καθηγητή Ε.Μ.Π, για την εμπιστοσύνη που μου έδειξε κατά την ανάθεση της παρούσας εργασίας.

Επίσης, θα ήθελα ιδιαιτέρως να ευχαριστήσω τον Διδακτορικό Ερευνητή κ. Βασίλειο Τσούτσουρα για την διαρκή καθοδήγηση που μου παρείχε, τις συμβουλές, τις παρατηρήσεις του και για την άμεση απόκριση του σε οποιαδήποτε απορία μου.

Τέλος, θα ήθελα να εκφράσω την απεριόριστη ευγνωμοσύνη μου στην οικογένεια μου για την ηθική και υλική υποστήριξη που μου παρείχε όλα τα χρόνια των σπουδών μου και για την βοήθεια της στην επίτευξη των στόχων μου. Αφιερώνω, λοιπόν, την παρούσα εργασία στους γονείς και τα αδέρφια μου.

Αθήνα, Φεβρουάριος 2018
Αντώνιος Ν. Κεκεμπάνος

Πίνακας περιεχομένων

1 *Τα λειτουργικά πραγματικού χρόνου RTOS στο Internet of Things*

1.1 Οι ανάγκες του IoT και τα RTOS

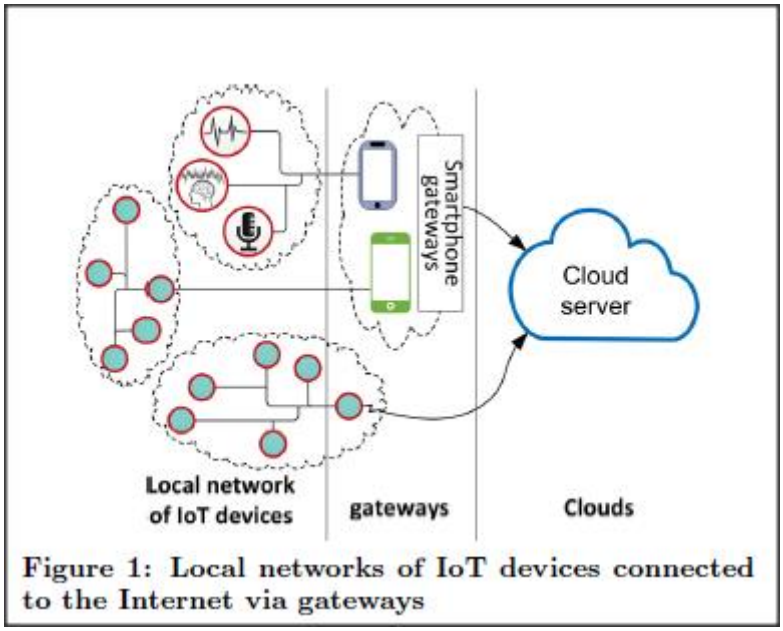


Figure 1: Local networks of IoT devices connected to the Internet via gateways

Σχήμα 1.1.1: Παράδειγμα state-of-the-art IoT εφαρμογής. [4].

Με τον όρο Internet of Things (IoT) εννοούμε το περιβάλλον στο οποίο ένα πλήθος μικρών ενσωματωμένων συστημάτων,από οικιακές συσκευές έως αυτοκίνητα και βιομηχανικούς αισθητήρες, είναι συνδεδεμένα στο Διαδίκτυο με σκοπό την βελτίωση της ποιότητας ζωής και της επιχειρησιακής παραγωγικότητας (efficiency).

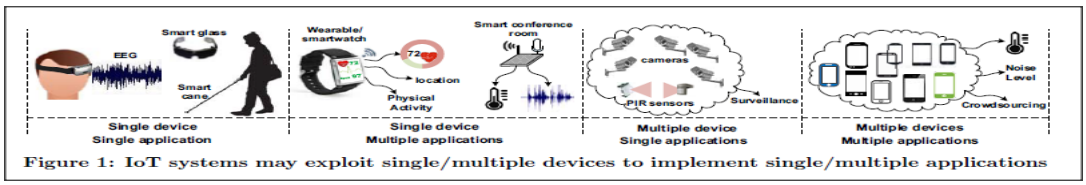


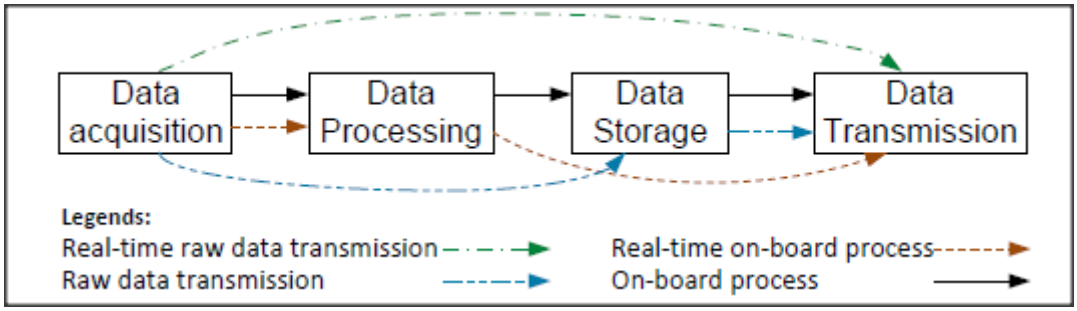
Figure 1: IoT systems may exploit single/multiple devices to implement single/multiple applications

Σχήμα 1.1.2: Παραδείγματα IoT εφαρμογών[4].

Οι εφαρμογές του IoT εκτείνονται σε πάρα πολλούς και διαφορετικούς τομείς όπως στην υγεία και τη διευκόλυνση της ζωής ηλικιωμένων και χρονίως πασχόντων, την αυτοματοποίηση των κατοικιών με στόχο την καλύτερη άνεση και μείωση της κατανάλωσης ενέργειας, την βελτίωση των συνθηκών ζωής στις πόλεις με εφαρμογές παρακολούθησης, διαχείρισης της οδικής κυκλοφορίας και της περιβαλλοντικής ρύπανσης οι οποίες λαμβάνουν δεδομένα από IoT συσκευές σε διάφορα σημεία. Στην βιομηχανία το IoT λαμβάνει την μορφή εφαρμογών ελέγχου, αυτοματοποίησης και παρακολούθησης διεργασιών που μπορούν να μειώσουν τα κόστη λειτουργίας και συντήρησης του εξοπλισμού. Στο σχήμα 1.1.2 βλέπουμε παραδείγματα IoT εφαρμογών με κατηγογοροποίηση που αφορά την πληθικότητα της αντιστοίχισης μεταξύ συσκευών-εφαρμογών, έτσι μια IoT εφαρμογή μπορεί να περιλαμβάνει μόνο μια συσκευή, όπως η εφαρμογή ανάλυσης και καταγραφής καρδιογραφήματος, ή ένα πλήθος συσκευών, όπως εφαρμογές παρακολούθησης (surveillance).

Ένας καθοριστικός παράγοντας που επέτρεψε την εκκίνηση της εξάπλωσης τέτοιους είδους συσκευών είναι η μείωση του κόστους τους. Σήμερα η αγορά υλικού (hardware) παρέχει μια τεράστια γκάμα υπολογιστικών συστημάτων σε μορφή ενός μόνο chip (System on Chip) με πολύ μικρό κόστος. Σε ένα τυπικό SoC των μερικών δολαρίων περιέχεται επεξεργαστής, μνήμη, περιφερειακά εισόδου - εξόδου, όπως μετατροπείς από αναλογικό σε ψηφιακό, και κυρίως υλικό ασύρματης δικτύωσης όπως για παράδειγμα Wifi, BLE (Bluetooth Low Energy), 3G, LTE, ZigBee, κάποιο LPWAN(Low Power Wide Area Network) όπως το LoRaWAN.

Αυτές οι IoT συσκευές ονομάζονται κόμβοι (nodes) και γενικά χαρακτηριστικά της λειτουργίας τους είναι η συλλογή δεδομένων από το περιβάλλον κι η αλληλεπίδραση με αυτό, η πιθανή επεξεργασία αυτών των δεδομένων, η ενδεχόμενη αποθήκευση μέρους αυτών για μελλοντική χρήση και κυρίως η αποστολή πληροφορίας στο διαδίκτυο και διασύνδεση μεταξύ τους. Η συνδεσιμότητα (connectivity) ενσύρματη ή ασύρματη είναι αυτή που ξεχωρίζει τις συσκεύες του IoT από τα γνωστά ενσωματωμένα συστήματα και είναι αυτή που επιτρέπει την αλληλεπίδραση των συσκευών αυτών τόσο μεταξύ τους όσο και με των άνθρωπο χωρίς να τίθενται χρονικοί η χωρικοί περιορισμοί. Στο σχήμα 1.1.1 η παράμετρος της συνδεσιμότητας γίνεται εμφανής αφού εδώ οι IoT κόμβοι στέλνουν τα δεδομένα που συλλέγουν σε απομακρυσμένη υπολογιστική υποδομή Cloud κι αυτό το κάνουν είτε άμεσα είτε έμμεσα μέσω μια πύλης IoT Gateway.



Σχήμα 1.1.3: Τα στάδια λειτουργίας μιας τυπικής IoT εφαρμογής [4].

Ταυτόχρονα με την απαίτηση εκτέλεσης των λειτουργιών που φαίνονται στο σχήμα 1.1.3, ένας IoT κόμβος υπόκειται και σε σημαντικούς περιορισμούς. Η διαθέσιμη υπολογιστική ισχύς είναι περιορισμένη όπως και το μέγεθος της μνήμης, λόγω φυσικού μεγέθους κυρίως, γεγονός που θέτει σημαντικές προκλήσεις στους σχεδιαστές. Ο πιο βασικός περιορισμός αφορά την κατανάλωση ενέργειας η οποία θα πρέπει είναι όσο το δυνατόν μικρότερη αν σκεφτεί κανείς ότι η πλειονότητα αυτών των συσκευών θα τροφοδοτείται από κάποιο συσσωρευτή. Αυτό σημαίνει ότι κάθε ένα από τα παραπάνω στάδια εκτέλεσης μιας IoT εφαρμογής θα πρέπει να εκτελείται αποδοτικά κάνοντας και αναγκαίους συμβιβασμούς εφόσον το επιτρέπουν οι περιορισμοί του προβλήματος. Για παράδειγμα η εφαρμογή μπορεί να μειώνει το ρυθμό αποστολής μετρήσεων αν αυτές παρουσιάζουν μικρή μεταβλητότητα ώστε να εξοικονομηθεί εύρος ζώνης (bandwidth) κι ενέργεια.

Επίσης υπάρχει η απαίτηση οι αναπτυσσόμενες εφαρμογές να είναι εύρωστες (robust) δηλαδή να έχουν την δυνατότητα ανάκαμψης από σφάλματα, όπως δικτύου, ενώ αρκετές από αυτές θα επιτελούν εργασίες στις οποίες τυχόν σφάλματα μπορεί να οδηγήσουν σε κίνδυνο εξοπλισμό και ανθρώπινη ζωή,όπως σε βιομηχανικές και ιατρικές εφαρμογές. Κάτι τέτοιο σημαίνει πως αν η ανάπτυξη του λογισμικού ενός νέου IoT προϊόντος ξεκινήσει από μηδενική βάση θα απαιτηθεί πολύ μεγάλο χρονικό διάστημα δοκιμών και νέων εκδόσεων με συνέπεια την σημαντική επιβάρυνση του λεγόμενου TTM (Time to Market) του υπό ανάπτυξη προϊόντος.

Ο παράγοντας TTM είναι μείζονος σημασίας για την αγορά του IoT. Σύμφωνα με προβλέψεις [3] η ανταγωνιστικότητα στο νέο αυτό τοπίο θα εξαρτάται από την δυνατότητα των εταιρειών να παρέχουν νέες καινοτόμες λειτουργίες είτε με νέα προϊόντα είτε με αναβαθμίσεις λογισμικού υπάρχοντος εξοπλισμού. Ο ανταγωνισμός θα έγκειται μεταξύ πολλών μικρών εταιρειών ,στην ουσία ομάδων ανάπτυξης των 5 έως 10 ατόμων, χωρίς όμως να λείπουν και μεγάλοι κατασκευαστές τεχνολογίας. Η ανάπτυξη ενός IoT προϊόντος δεν απαιτεί την ύπαρξη κάποιας σημαντικής υποδομής ή μεγάλο κεφάλαιο οπότε δίνεται η δυνατότητα σε ομάδες σχεδιαστών να παράξουν μόνοι τους ένα προϊόν. Την ίδια στιγμή η μεγάλη μεταβλητότητα των συνθηκών σε συνδυασμό με τον ισχυρό ανταγωνισμό ελαχιστοποιούν το χρονικό παράθυρο στο οποίο μπορεί ένα προϊόν να βγει στην αγορά και να είναι επικερδές. Οπότε το λεγόμενο TTM στην αγορά αυτή παίζει σημαντικότατο ρόλο αν συνδυάζεται, βέβαια, με υψηλή ποιότητα του τελικού προϊόντος.

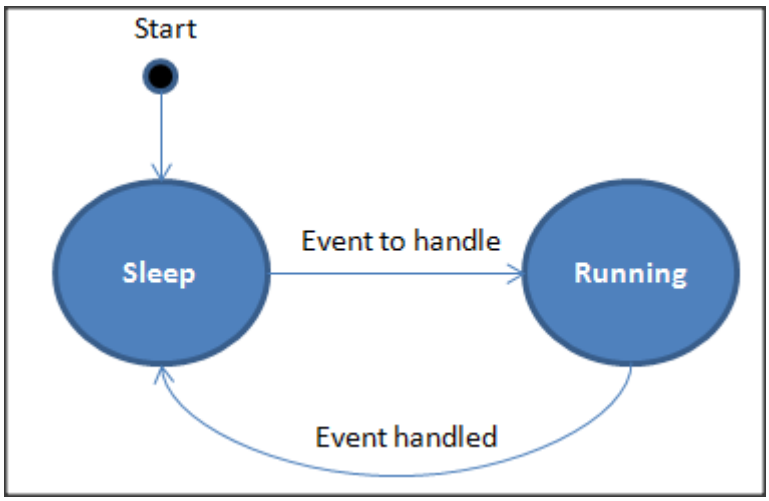
Επιπρόσθετα είναι συνηθισμένο, κι αναγκαίο, μια εταιρεία να έχει μια γκάμα προϊόντων ώστε να μπορεί να εξυπηρετήσει ένα σύνολο διαφορετικών αναγκών των πελατών της ή για να είναι σε θέση να προσφέρει ολοκληρωμένες λύσεις. Αυτό στο πεδίο του IoT μεταφράζεται σε προϊόντα με πιθανά διαφορετικές αρχιτεκτονικές μικροεπεξεργαστών όπως για παράδειγμα σε ένα IoT node κι ένα IoT gateway. Αν η ανάπτυξη του λογισμικού ξεκινήσει από μηδενική βάση, χωρίς δηλαδή κάποιου είδους framework, δεν θα υπάρχει κάποιο όφελος από την ανάπτυξη της μιας εφαρμογής για την ανάπτυξη της άλλης. Αν όμως και οι δύο βασίζονταν σε ένα όμοιο, από άποψη API,αλλά φορητό framework τότε σίγουρα η

επαναχρησιμοποίηση κώδικα θα ήταν ευκολότερη ενώ θα μειωνόταν το TTM επόμενων προϊόντων.

Η χρήση ενός λειτουργικού συστήματος πραγματικού χρόνου RTOS (Real Time Operating System) μπορεί να προσφέρει λύση σε όλες τις παραπάνω απαιτήσεις. Ξεκινώντας από τις στάδια της εκτέλεσης μιας IoT εφαρμογής , όπως καταγράφονται στο σχήμα 1.1.3, η χρήση ενός RTOS μπορεί να διευκολύνει την ανάπτυξη αφού ο κώδικας για κάθε ένα από τα παραπάνω στάδια μπορεί να γραφεί ανεξάρτητα. Στο τελικό σύστημα μπορούν να συνδυαστούν σαν διεργασίες (processes) του λειτουργικού που θα χρησιμοποιούν τους μηχανισμούς διαδιεργασιακής επικοινωνίας και συγχρονισμού που θα παρέχει το λειτουργικό. Επίσης είναι αναγκαίο το σύστημα να αποκρίνεται γρήγορα σε εξωτερικά event να είναι δηλαδή real time όπως συμβαίνει για παράδειγμα σε βιομηχανικές εφαρμογές ελέγχου. Εδώ ακριβώς εισάγεται η απαίτηση το λειτουργικό σύστημα να είναι πραγματικού χρόνου κι όχι γενικού σκοπού. Ακόμα κι όταν η εφαρμογή δεν είναι αυστηρά πραγματικού χρόνου η χρήση ενός RTOS είναι καλύτερη επιλογή καθώς δίνει εγγυήσεις για την αξιοπιστία και την αποκρισιμότητα του συστήματος.

Επίσης, όπως αναφέρθηκε η πλειονότητα των IoT κόμβων είναι περιορισμένοι από άποψη υπολογιστικών πόρων και επεξεργαστικής ισχύος κάτι που παλιότερα θα έκανε την χρήση ενός λειτουργικού απαγορευτική. Τα παρεχόμενα όμως σήμερα λειτουργικά πραγματικού χρόνου έχουν αναπτυχθεί με γνώμονα αυτούς τους περιορισμούς κι αυτό έχει ως αποτέλεσμα να είναι η δυνατή η χρήση τους σε πάρα πολλές αρχιτεκτονικές και πλατφόρμες ανάπτυξης παρέχοντας ευελιξία στην επιλογή του υλικού.

Ακόμη, μια πολύ σημαντική απαίτηση που μπορεί να καλυφθεί με την χρήση ενός RTOS είναι ο περιορισμός της κατανάλωσης ενέργειας. Με την χρήση ενός τέτοιου ο σχεδιαστής μπορεί να αναθέσει διαφορετικές προτεραιότητες στις διάφορες διεργασίες ανάλογα με την εγγύηση που απαιτεί η εφαρμογή για το χρόνο εκτέλεσης τους. Επιπλέον χρησιμοποιώντας μηχανισμούς συγχρονισμού μπορεί σηματοδοτηθεί η εκτέλεση κάποια ενέργειας από την ανίχνευση κάποιου γεγονότος. Έχει δηλαδή ο σχεδιαστής την δυνατότητα εύκολα να κάνει το σύστημα οδηγούμενο από γεγονότα (event-driven),σχήμα 1.1.5. Κάτι τέτοιο οδηγεί σε σημαντικό περιορισμό της δαπανούμενης ενέργειας από την συσκευή καθώς στο χρόνο που δεν συμβαίνουν γεγονότα το σύστημα μπορεί να βρίσκεται σε κατάσταση αδρανείας(idle). Τέλος, πολλά διαθέσιμα RTOS ενσωματώνουν λειτουργίες χαμηλής κατανάλωσης ενέργειας που μπορούν να χρησιμοποιηθούν αν κρίνεται απαραίτητο,με κάποιο κόστος, ωστόσο, στο σύνολο των διαθέσιμων λειτουργιών από την πλευρά του λειτουργικού.



Σχήμα 1.1.5 : Παράδειγμα event-driven λειτουργίας.

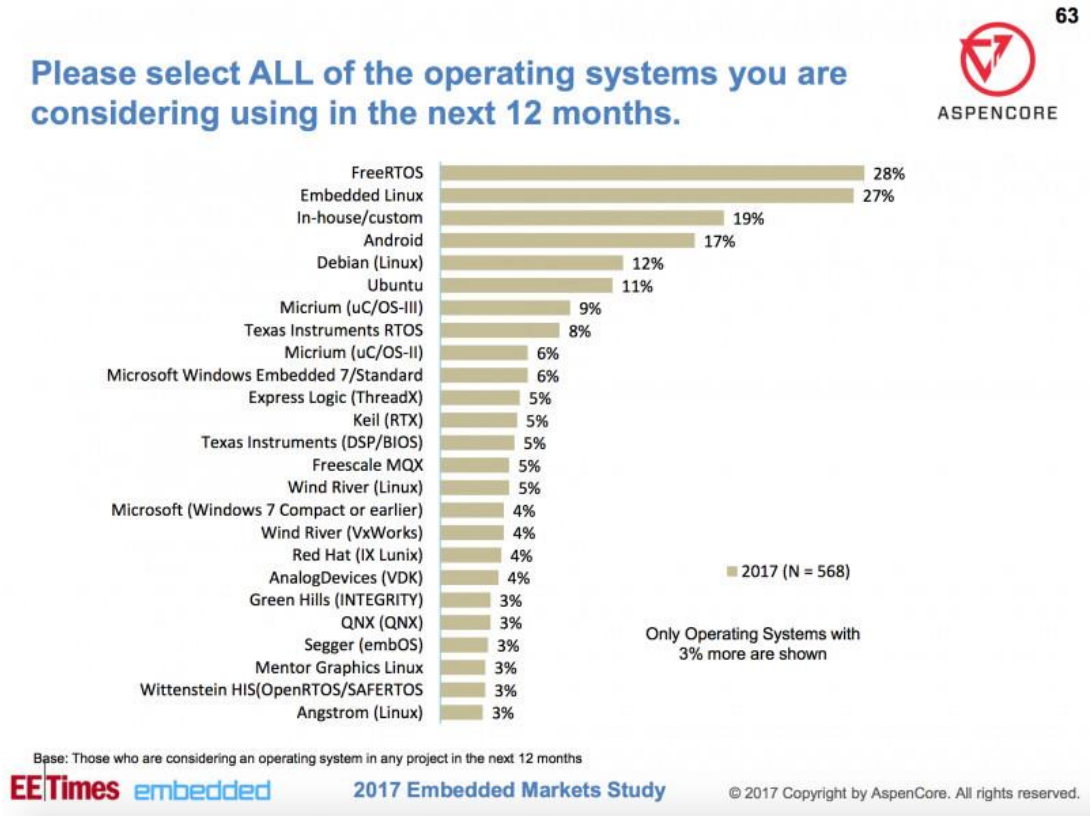
Επιπλέον,πολλά RTOS παρέχουν έτοιμες δυνατότητες δικτύωσης,όπως Wifi drivers TCP/IP stack, Bluetooth, LoRaWan κι άλλα. Επομένως, ο σχεδιαστής απαλλάσσεται από το κόστος ανάπτυξης λογισμικού δικτύωσης κι είναι σε θέση να χρησιμοποιήσει ένα έτοιμο software stack δοκιμασμένο κι αξιόπιστο με υποστήριξη εκτενούς τεκμηρίωσης του παρεχόμενου API. Κάτι τέτοιο έχει πολύ μεγάλο αντίκτυπο και μπορεί να περιορίσει το TTM σε IoT λύσεις που η συνδεσιμότητα είναι αναγκαίο χαρακτηριστικό.

Όπως αναφέρθηκε στην αγορά του IoT υπάρχει η ανάγκη γρήγορης ενσωμάτωσης νέων καινοτόμων δυνατοτήτων σε προϊόντα που θα πρέπει παράλληλα να είναι αξιόπιστα και να παρέχουν εγγυήσεις για την ασφάλεια (safety) όταν η εφαρμογή το απαιτεί. Η χρήση ενός λειτουργικού μπορεί να συμβάλλει άμεσα σε αυτό καθώς θα αποτελέσει ένα αξιόπιστο και εκτενώς δοκιμασμένο framework πάνω στο οποίο μπορεί να υλοποιηθούν οι ιδιαίτερες δυνατότητες του υπό σχεδίαση προϊόντος. Ο πυρήνας δηλαδή της εφαρμογής, το RTOS, ο οποίος περιλαμβάνει όλες τις βασικές λειτουργίες,όπως η σύνδεση στο διαδίκτυο,δεν χρειάζεται να να αναπτυχθεί ξανά. Ταυτόχρονα δίνονται εγγυήσεις για την αξιόπιστη κι απροβλημάτιστη λειτουργία του σύμφωνα με τις προδιαγραφές που δίνει το λειτουργικό. Ιδιαίτερα όσο αφορά την ασφάλεια(safety) υπάρχουν και εκδόσεις RTOS που φέρουν πιστοποιήσεις από τρίτους οργανισμούς κι είναι κατάλληλες για τέτοιες εφαρμογές με υψηλές απαιτήσεις στο τομέα αυτό,όπως το SAFERTOS, σχήμα 1.1.6. Συνεπώς, ο σχεδιασμός του προϊόντος μπορεί να εστιάσει εξαρχής στην προστιθέμενη αξία του βασιζόμενος σε γερά θεμέλια.

Σχήμα 1.1.6 : Το λειτουργικό SAFERTOS με πιστοποίηση TUV.

Ακόμα,μια εταιρεία είναι συνηθισμένο να έχει γκάμα προϊόντων με διαφοροποιήσεις στην αρχιτεκτονική μικροεπεξεργαστή που χρησιμοποιεί το κάθε ένα. Τα περισσότερα RTOS που κυκλοφορούν μπορούν να χρησιμοποιηθούν σε ένα πολύ μεγάλο εύρος αρχιτεκτονικών από 8 μέχρι 32 και 64 bit. Συνεπώς δίνεται η δυνατότητα χρήσης του λειτουργικού σε ανομοιογενείς μεταξύ τους συσκευές χωρίς να αλλάζει σημαντικά το παρεχόμενο API, για όσες βέβαια δυνατότητες του λειτουργικού μπορεί να υποστηρίξει η υποκείμενη αρχιτεκτονική. Έτσι δίνεται η δυνατότητα η υπάρχουσα γνώση που έχει προκύψει από την ανάπτυξη παλιότερων προϊόντων πάνω στο ίδιο λειτουργικό να αξιοποιηθεί ως επίσης και επαναχρησιμοποίησης υπάρχοντος κώδικας. Αυτή η παράμετρος μειώνει το TTM για επόμενα προϊόντα της ίδιας εταιρείας.

Τέλος δεν θα πρέπει να αγνοηθούν κι άλλες έτοιμες λειτουργίες που μπορεί να παρέχει η χρήση ενός RTOS που απευθύνεται σε IoT συσκευές. Όπως για παράδειγμα σουίτες κρυπτογράφησης η οποία αποτελεί ένα σημαντικότατο εργαλείο για την παροχή ασφάλειας (security) και στο IoT, εφόσον η εκτεταμένη συνδεσιμότητα των συσκευών συνεπάγεται και μεγαλύτερο κίνδυνο. Μια άλλη λειτουργία που υποστηρίζουν αρκετά προϊόντα είναι η δυνατότητα OTA αναβάθμισης (Over The Air) απομακρυσμένων συσκευών που είναι συνδεδεμένες στο διαδίκτυο. Αυτή η δυνατότητα είναι αναγκαία στο κόσμο του IoT όπου το πλήθος των συσκευών και η χωρική διασπορά τους κάνουν ασύμφορη την ενημέρωση του λογισμικού τους με άλλο τρόπο. Μια άλλη λειτουργία που συνήθως παρέχεται είναι ένα σύστημα αρχείων για την αξιοποίηση ενός φυσικού μέσου αποθήκευσης, όπως SD κάρτα. Γενικά, για κάθε λειτουργικό ,παράλληλα με την βασική ανάπτυξη του, αναπτύσσονται και επιπλέον δυνατότητες ώστε να παρέχει εργαλεία και διευκολύνσεις στον σχεδιαστή που θα το χρησιμοποιήσει. Ο στόχος βέβαια είναι να γίνει πιο δελεαστική η επιλογή του μέσα σε ένα εξαιρετικά ανταγωνιστικό περιβάλλον και για τα ίδια τα λειτουργικά ενσωματωμένων συστημάτων,όπως αποτυπώνεται στο σχήμα 1.1.7.

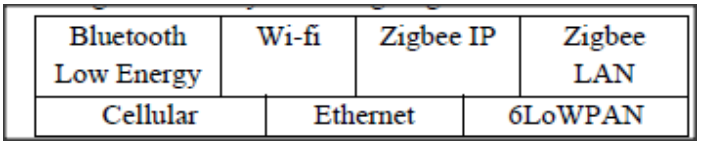


Σχήμα 1.1.7 : Το τοπίο χρήση λειτουργικών στα ενσωματωμένα συστήματα.

Όπως βλέπουμε, η επιλογή της χρήσης του κατάλληλου RTOS μπορεί να μειώσει σημαντικά το TTM και να ωθήσει σε ανώτερα επίπεδα την ποιότητα των εφαρμογών. Προφανώς υπάρχει το κόστος της εκμάθησης ενός λειτουργικού αλλά το κόστος αυτό είναι εφάπαξ και ο χρόνος χρήσης του λειτουργικού αρκετά χρόνια. Υπάρχουν ακόμα, και λειτουργικά τα οποία έχουν άδεια χρήσης επί πληρωμή. Στην περίπτωση αυτή όμως παρέχονται επιπλέον διευκολύνσεις όπως 24/7 τεχνική υποστήριξη που μπορεί να οδηγήσει σε γρήγορη απόσβεση του παραπάνω κόστους λόγω και πάλι του μικρότερου TTM αφού τα όποια προβλήματα θα αντιμετωπίζονται ταχύτερα.

1.2 Σχετικές εργασίες

Η χρήση λειτουργικού συστήματος RTOS στο IoT έχει μελετηθεί κι υπάρχουν παραδείγματα στην βιβλιογραφία Ακολουθεί η συνοπτική περιγραφή μερικών σχετικών εργασιών και δημοσιεύσεων. Η πρώτη εργασία [1] αποτελεί μια μικρή έρευνα για την χρήση λειτουργικού σε περιβάλλον IoT με σύντομη περιγραφή των λειτουργικών ARM mbed, RIOT, Contiki, TinyOS, Nano-RK και FreeRTOS. Για κάθε ένα από αυτά αναφέρονται οι ιδιαίτερες δυνατότητες του, οι αρχιτεκτονικές μικροεπεξεργαστών στις οποίες μπορεί να χρησιμοποιηθεί, το περιβάλλον ανάπτυξης και η γλώσσα προγραμματισμού που χρησιμοποιείται. Γίνεται ιδιαίτερη αναφορά στις ενσωματωμένες δυνατότητες δικτύωσης του κάθε λειτουργικού όπως φαίνεται στο σχήμα 1.2.1:



Σχήμα 1.2.1 : Τεχνολογίες δικτύωσης στο ARM mbed OS[1].

Η δικτύωση και η ασφάλεια αναφέρονται ως οι βασικοί πυλώνες του IoT με τεχνικές κρυπτογράφησης κι απόκρυψης δεδομένων να είναι απαραίτητες για την αντιμετώπιση επιθέσεων και προστασία των περιορισμένων υπολογιστικών πόρων των IoT συσκευών. Μετά την σύντομη παρουσίαση των RTOS, καταλήγει πως τα περισσότερα από αυτά είναι εξοπλισμένα με τα σημαντικότερα πρωτόκολλα δικτύωσης κι επικοινωνίας καθώς και με παραμέτρους ασφαλείας ενώ είναι βελτιστοποιημένα για την αποδοτική αξιοποίηση των περιορισμένων υπολογιστικών πόρων. Τέλος, επισημαίνεται ότι η χρήση RTOS θα κάνει την IoT υποδομή πιο αξιόπιστη κι ανθεκτική στις όποιες επιθέσεις.

Η επόμενη εργασία που μελετήθηκε [2] έχει ως στόχο να δώσει κάποιες κατευθυντήριες γραμμές για την επιλογή του κατάλληλου RTOS για την εκάστοτε IoT πλατφόρμα. Αρχικά αναφέρει την χρήση ενός RTOS ως αναγκαίο εργαλείο για την αντιμετώπιση των πολλαπλών προκλήσεων του IoT. Μάλιστα γίνεται κι ένας διαχωρισμός μεταξύ βιομηχανικών κι καταναλωτικών (consumer) IoT εφαρμογών, αναφέροντας ότι στις δεύτερες υπάρχει η ανάγκη για κόμβους πολύ πλουσιότερους σε λειτουργίες γεγονός που επηρεάζει άμεσα την IoT πλατφόρμα που θα χρησιμοποιηθεί (μικροεπεξεργαστής) όπως και το λειτουργικό.

Στην συνέχεια επισημαίνονται τα βασικότερα χαρακτηριστικά ενός λειτουργικού τα οποία είναι η δομή του πυρήνα (monolithic,layered ή microkernel),ο αλγόριθμος δρομολόγησης διεργασιών, που έχει και την μεγαλύτερη επίδραση στο κατά πόσο το σύστημα είναι πραγματικού χρόνου, και το μοντέλο προγραμματισμού.Σε ένα μοντέλο προγραμματισμού, για παράδειγμα, μπορεί να όλες οι διεργασίες να έχουν πρόσβαση σε κοινή μνήμη ενώ σε άλλο η μνήμη μεταξύ των διεργασιών να είναι διαχωρισμένη. Ακολουθεί, κι εδώ, μια συνοπτική παρουσίαση των λειτουργικών Contiki, Tiny OS, FreeRTOS ,ChibiOS/RT, Erika

Enterprise και RIOT ως επίσης και μια συνοπτική σύγκριση που παρουσιάζεται στο σχήμα 1.2.2.

TABLE 1. KEY CHARACTERISTICS OF TINYOS, CONTIKI, RIOT, AND LINUX. (✓) FULL SUPPORT, (○) PARTIAL SUPPORT, (X) NO SUPPORT.

OS	min RAM	min ROM	C support	C++ support
TinyOS	< 1kB	< 4kB	x	x
Contiki	< 2kB	< 30kB	○	x
RIOT	~ 1.5kB	~ 5kB	✓	✓
Linux	~ 1MB	~ 1MB	✓	✓

OS	multi-threading	MCU w/o MMU	modularity	real-time
Tiny OS	○	✓	x	x
Contiki	○	✓	○	○
RIOT	✓	✓	✓	✓
Linux	✓	x	○	○

Σχήμα 1.2.2 : Πίνακας σύγκρισης λειτουργικών για ενσωματωμένα συστήματα[2].

Η εργασία εστιάζει στο RIOT (Real-time operating system for IoT) του οποίου χαρακτηριστικά είναι η ελαχιστοποίηση της κατανάλωσης ενέργειας κι αποτυπώματος μνήμης καθώς και η παροχή ενός κοινού API ανεξάρτητα από την υποκείμενη αρχιτεκτονική. Τέλος περιγράφεται η διαδικασία πειραματικής αξιολόγησης του σε μια διάταξη 8 ασύρματων κόμβων (nodes) που επικοινωνούν κι αποστέλλουν δεδομένα αναλογικών μετρήσεων μέσω Wifi σε ένα συγκεκριμένο κόμβο. Οι κόμβοι υλοποιήθηκαν στην αναπτυξιακή πλακέτα EasyMx PRO v7 for STM32 με χρήση ARM-Cortex M4,οι οποίοι εκτελούσαν το RIOT κι ο γενικός στόχος ήταν η εξερεύνηση των δυνατοτήτων του λειτουργικού.

Η τρίτη εργασία που αφορά την χρήση RTOS στο IoT [3] κάνει μια γενική περιγραφή του IoT περιβάλλοντος ,σχήμα 1.2.3, κι εστάζει στα χαρακτηριστικά που θα πρέπει να ικανοποιεί ένα RTOS ώστε να είναι κατάλληλο για το πεδίο αυτό, επισημαίνοντας παράλληλα την αναγκαιότητα της χρήσης του.

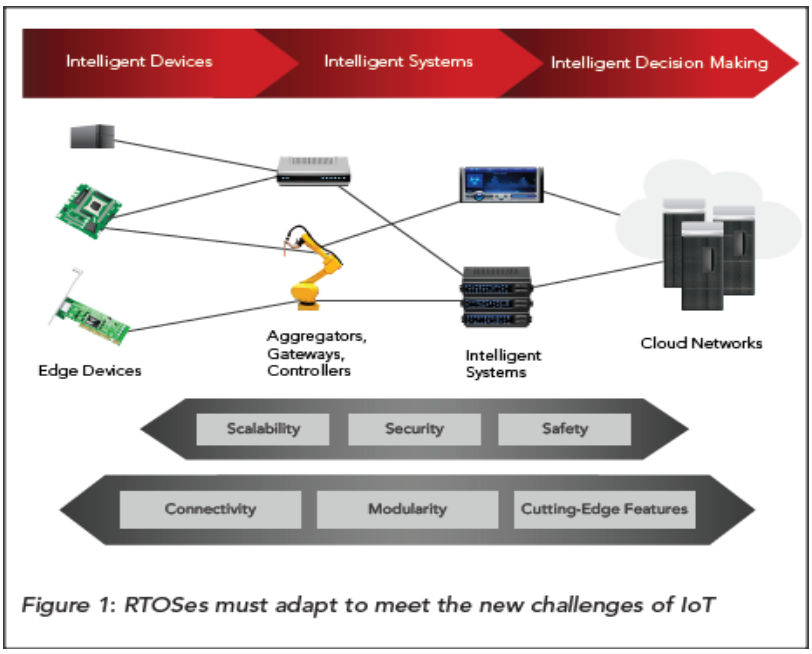
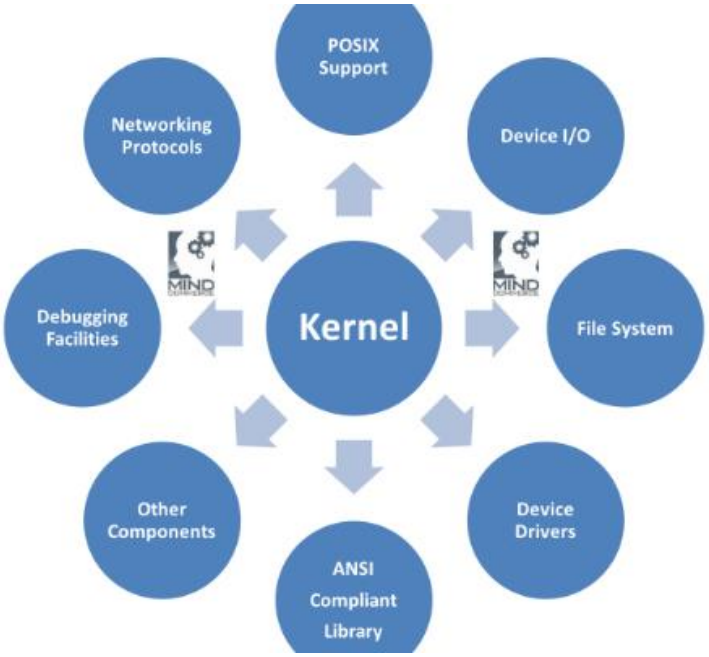


Figure 1: RTOSes must adapt to meet the new challenges of IoT

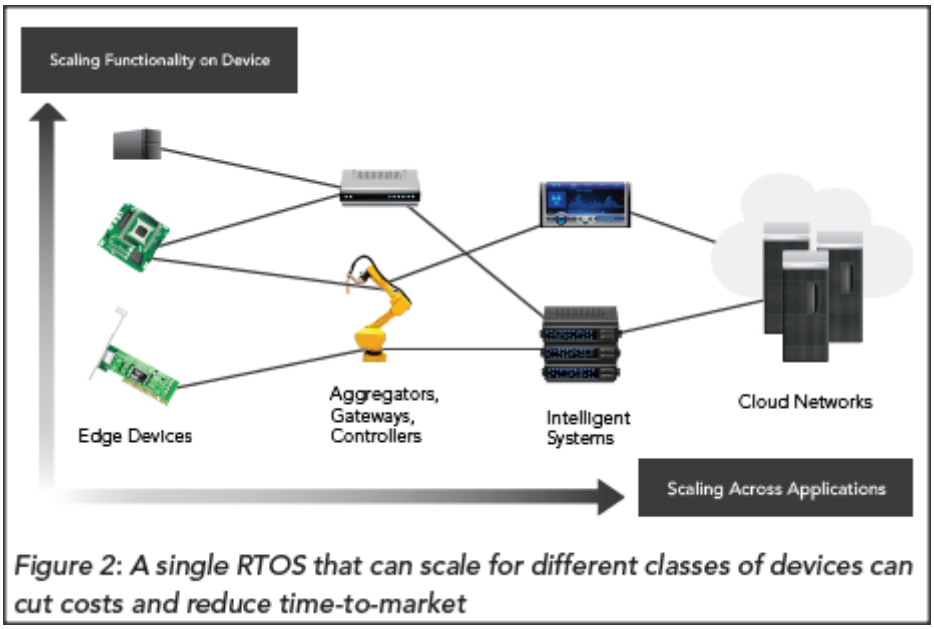
Σχήμα 1.2.3 :Μια τυπική IoT υποδομή[3].

Ένα IoT RTOS πρέπει να έχει σπονδυλωτή (modular) δομή δηλαδή ο πυρήνας του να είναι διαχωρισμένος από τις επιπρόσθετες λειτουργίες,όπως networking, έτσι ώστε μελλοντικές αλλαγές και προσθήκες να είναι δυνατές χωρίς μεγάλο κόστος,σχήμα 1.2.4.



Σχήμα 1.2.4 :Αρθρωτή(modular) δομή ενός IoT RTOS.

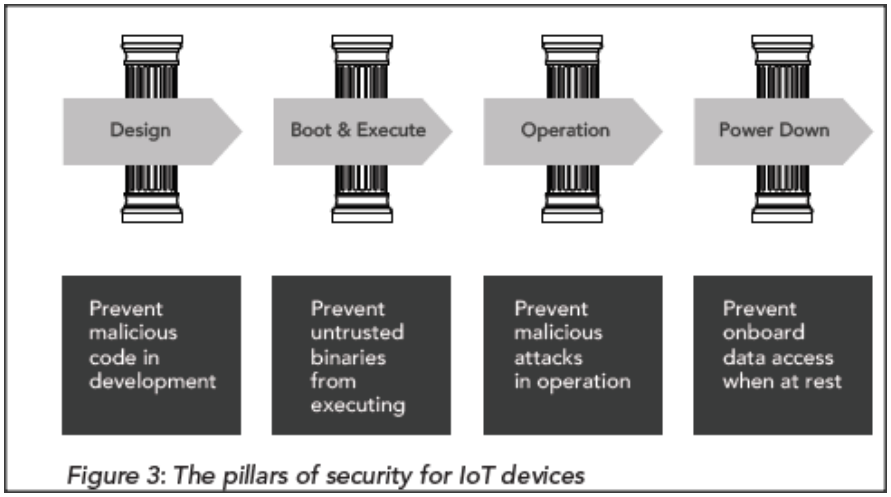
Επίσης θα πρέπει να χαρακτηρίζεται από επεκτασιμότητα (scalability) ώστε να μπορεί να χρησιμοποιηθεί σε ανομοιογενείς πλατφόρμες,σχήμα 1.2.5.



Σχήμα 1.2.5 :Η επεκτασιμότητα(scalability) ενός RTOS μπορεί να μειώσει τα κόστη και το TTM στο IoT [3].

Ακόμη,σύμφωνα με την εργασία, είναι αναγκαίο ένα RTOS για IOT να υποστηρίζει λειτουργίες ασφάλειας (security),με τους βασικούς πυλώνες του security στο σχήμα 1.2.6, ενσωματωμένες στην αρχιτεκτονική του πυρήνα του λειτουργικού καθώς η προσθήκη

τέτοιων λειτουργιών στο χώρο χρήστη είναι δαπανηρή,αναποτελεσματική κι ενέχει κινδύνους.



Σχήμα 1.2.6 :Οι πυλώνες της ασφάλειας των IoT συσκευών.

Επίσης, τονίζεται ότι η ασφάλεια(safety) και η συνδεσιμότητα (connectivity) αποτελούν πολύ βασικές παραμέτρους για τις οποίες πρέπει να έχει ληφθεί μέριμνα κατά την ανάπτυξη του λειτουργικού. Ο παράγοντας της ασφάλειας (safety) είναι αυτός που επιβάλλει ενός λειτουργικού πραγματικού χρόνου (real-time). Τέλος προτείνεται η χρήση του VxWorks IoT RTOS το οποίο είναι εμπορικό προϊόν με κόστος άδειας χρήσης, σχήμα 1.2.7.

Σχήμα 1.2.7 :Το VxWorks IoT RTOS της Wind River (θυγατρική της Intel).

Συμπερασματικά,όπως φαίνεται από τις παραπάνω εργασίες η χρήση ενός RTOS το οποίο είναι σε θέση να καλύψει τις ανάγκες του IoT αποτελεί σχεδόν μονόδρομο για την ανάπτυξη ενός ανταγωνιστικού IoT προϊόντος το οποίο θα είναι αξιόπιστο, ασφαλές κι εύρωστο ενώ παράλληλα θα έχει την δυνατότητα να εξελιχθεί ώστε να καλύπτει και μελλοντικές ανάγκες(future proof) χωρίς σημαντικά κόστη.

2 *Το λειτουργικό πραγματικού χρόνου FreeRTOS*

2.1 Εισαγωγή στο FreeRTOS

Το FreeRTOS είναι ένα δημοφιλές λειτουργικό σύστημα πραγματικού χρόνου για χρήση σε ενσωματωμένα συστήματα το οποίο είναι αρκετά συμπαγές κι ελαφρύ ώστε να μπορεί να χρησιμοποιείται σε αρχιτεκτονικές περιορισμένες σε πόρους μνήμης κι επεξεργαστικής ισχύος.Ο δημιουργός του project είναι ο Richard Barry ο οποίος διετέλεσε επικεφαλής της Real Time Engineers Ltd. στην οποία άνηκε μέχρι πρότινος το FreeRTOS project και ήταν υπεύθυνη για την ανάπτυξη του.Με τον όρο πραγματικού χρόνου εννοούμε ότι δίνεται η δυνατότητα της εγγύησης του μέγιστου χρόνου που απαιτείται ώστε το σύστημα να αντιδράσει σε ένα,εξωτερικό συνήθως, γεγονός, χωρίς την απαίτηση όμως το σύστημα να παραμένει αδρανές το υπόλοιπο διάστημα.

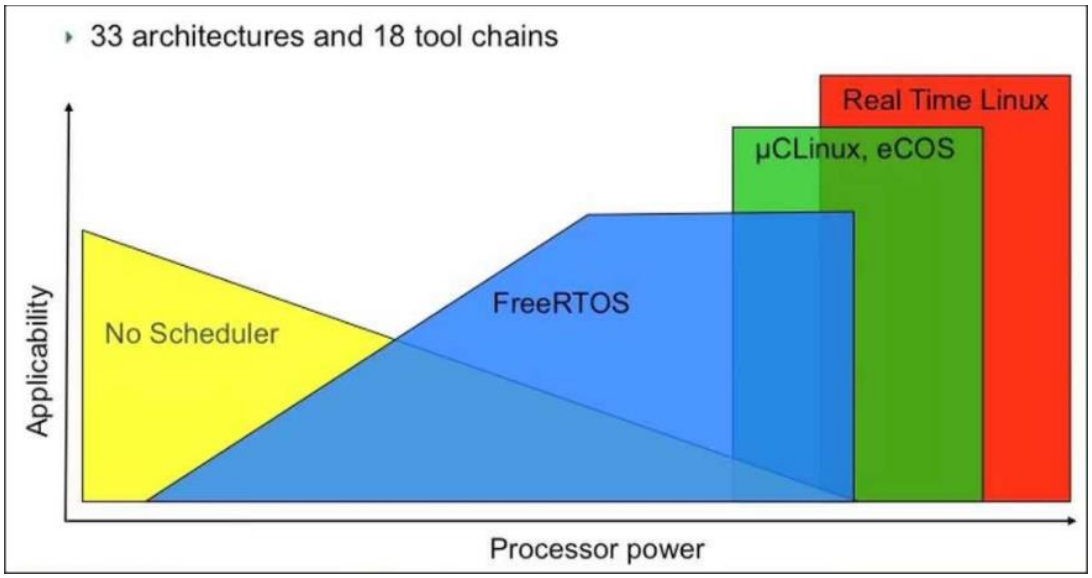
Το FreeRTOS είναι αρκετά δημοφιλές και βασικοί λόγοι για την διάδοση του είναι η ποιότητα του που διασφαλίζεται από την συνεχή ανάπτυξη του, και η συνεργασία της ομάδας ανάπτυξης με μερικούς από τους μεγαλύτερους κατασκευαστές hardware όπως φαίνεται στο σχήμα 2.1.1.

Σχήμα 2.1.1 :Συνεργάτες FreeRTOS.

Επίσης είναι ανοικτού κώδικα (open source) χωρίς η άδεια διανομής του να απαγορεύει την χρήση του σε εμπορικές εφαρμογές με κλειστό κώδικα. Αυτό έχει ως συνέπεια,σήμερα να θεωρείται ως de facto standard σε ενσωματωμένες εφαρμογές πραγματικού χρόνου και να χρησιμοποιείται σε χιλιάδες προϊόντα,σχήμα 2.1.2. Μερικά παραδείγματα των αρχιτεκτονικών που έχει γίνει port είναι: ARM Cortex A9(το Zynq-7000 SoC της Xilinx περιλαμβάνει δύο τέτοια cores),ARM Cortex-M3(το core που περιλαμβάνει το Particle Photon),ARM Cortex-A,Atmel AVR,Intel x86.

Η γλώσσα στην οποία έχει γραφεί το FreeRTOS είναι η C ενώ η προσπάθεια για απλότητα και μικρό μέγεθος εξηγεί το γεγονός ότι ο ίδιος ο πυρήνας του (kernel) αποτελείται από μόλις 3 αρχεία C (tasks.c, queue.c και list.c).Τα παραπάνω αρχεία ,που είναι κοινά για κάθε αρχιτεκτονική, καθώς κι ένα ακόμα αρχείο,ειδικό για την εκάστοτε αρχιτεκτονική, αποτελούν το σύνολο του πηγαίου κώδικα που απαιτείται για να χρησιμοποιηθεί το FreeRTOS. Παράλληλα το παρεχόμενο API είναι αρκετά εύκολο στην κατανόηση αλλά και στην χρήση του χάρη στο εκτενές documentation.

Ένα χαρακτηριστικό παράδειγμα για τις απαιτήσεις μνήμης είναι το port RL78 (16-bit CPU core by Renesas Electronics) που υποστηρίζει 13 tasks,2 queues και 4 software timers με memory footprint < 4K bytes RAM. Τα tasks, queues, software timers αποτελούν σημαντικά concepts που θα εξηγηθούν στο κεφάλαιο με την περιγραφή του FreeRTOS.



Σχήμα 2.1.2 :Η θέση του FreeRTOS στην αγορά σήμερα.

Πέρα από την βασική έκδοση του πυρήνα προσφέρονται κι άλλες παραλλαγές με επιπλέον λειτουργικότητα ή κάποια πιστοποίηση. Όπως για παράδειγμα το SAFERTOS που φέρει TÜV πιστοποίηση για την λειτουργία του,ενώ παράλληλα προσφέρεται και τεχνική

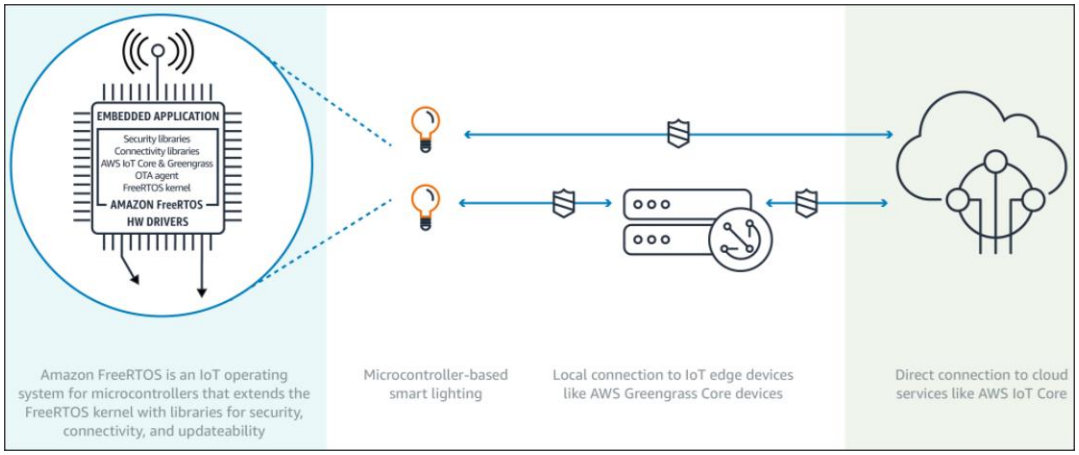
υποστήριξη. Άλλη εμπλουτισμένη έκδοση που παρέχεται είναι η FreeRTOS+TCP το οποίο περιέχει κώδικα για την πλήρη υποστήριξη TCP/IP stack ως επίσης και η έκδοση FreeRTOS+FAT για την υποστήριξη συστήματος αρχείων. Τα παραπάνω μπορούν να συνδυαστούν με αποτέλεσμα ο σχεδιαστής ενός συστήματος να έχει εξαρχής στα χέρια του ένα framework υψηλής ποιότητας.

Τα παραπάνω πλεονεκτήματα του FreeRTOS και ιδιαίτερα η διάδοση του και η καθιέρωση του ως παγκόσμιο πρότυπο της βιομηχανίας κάνουν την ενασχόληση κι εκμάθηση πολύ δελεαστικές.Θα πρέπει να αναφερθεί,ότι στην διάρκεια εκπόνησης αυτής της εργασίας η εποπτεία και διαχείριση του FreeRTOS πέρασε στην Amazon Web Services(AWS) με τον Richard Barry να εργάζεται σαν μέλος της νέας ομάδας ανάπτυξης.

Σχήμα 2.1.3 :Το λογότυπο της Amazon Web Services.

Σχήμα 2.1.4 :Το λογότυπου του Amazon FreeRTOS.

Η Amazon Web Services(AWS),σχήμα 2.1.3, προσφέρει μια πλατφόρμα υπηρεσιών νέφους(cloud) χαμηλού κόστους κι υψηλής αξιοπιστίας για την υποστήριξη χιλιάδων εταιριών σε 190 χώρες. Το 2015 η AWS προσέθεσε δυνατότητες του λεγόμενου Internet of Things (IoT) και τώρα προσφέρει το Amazon FreeRTOS, σχήμα 2.1.4, για την εύκολη σύνδεση συσκευών στο cloud. Χρησιμοποιείται ο πυρήνας του FreeRTOS κι έχουν προστεθεί βιβλιοθήκες που διευκολύνουν την ανάπτυξη εφαρμογών σε χαμηλής ισχύος(low power) συσκευές παρέχοντας δυνατότητες ασφάλειας (security), διασύνδεσης και διαχείρισης.Το σχήμα 2.1.5 δείχνει ένα παράδειγμα εφαρμογής πάνω στο Amazon FreeRTOS.



Σχήμα 2.1.5 :Παράδειγμα εφαρμογής έξυπνου φωτισμού βασιζόμενη στο Amazon FreeRTOS.

2.2 Βασικές έννοιες

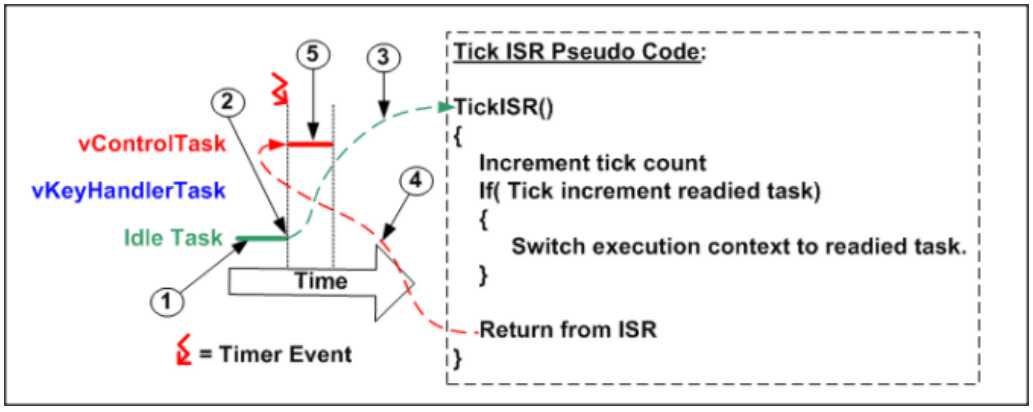
2.2.1 Διεργασίες FreeRTOS Tasks

Τα tasks του FreeRTOS είναι αντίστοιχα με τα processes του Linux. Αποτελούν δηλαδή ξεχωριστές οντότητες με τον δικό τους κώδικα και μνήμη. Θα πρέπει να αναφερθεί ότι δεν έχουμε μηχανισμούς εικονικής μνήμης όπως σε ανώτερα λειτουργικά (Linux Windows) και κατά συνέπεια δεν υπάρχει αυστηρή προστασία και διαχωρισμός των περιοχών μνήμης στην οποία κάθε task έχει πρόσβαση. Αυτό που πράγματι είναι υπεύθυνο το FreeRTOS όσο αφορά την μνήμη, είναι ότι εκχωρεί σε κάθε task ένα βάθος στοίβας το οποίο δηλώνεται στην κλήση δημιουργίας του task και είναι το μέγιστο που μπορεί να χρησιμοποιήσει κατά την διάρκεια της ζωής του. Συνεπώς θα πρέπει εκ των προτέρων ο προγραμματιστής να κάνει μια εκτίμηση για το μέγεθος της μνήμης που θα χρησιμοποιηθεί καθώς αυτή δεσμεύεται την στιγμή της δημιουργίας του task. Έτσι αν το μέγεθος αυτό δεν είναι ικανό θα έχουμε stack overflow ενώ αν είναι μεγαλύτερο από όσο χρειάζεται, σπατάλη μνήμης. Γενικά το stack overflow είναι βασική πηγή αστάθειας των εφαρμογών ενώ παρέχονται κι εργαλεία για την ανίχνευση του από τον πυρήνα του FreeRTOS. Συγκεκριμένα στο αρχείο *FreeRTOSConfig.h* υπάρχει η επιλογή ***configCHECK_FOR_STACK_OVERFLOW configuration constant*** και στην περίπτωση που αυτό έχει τεθεί στη τιμή 1 κι ανιχνευθεί overflow καλείται η callback συνάρτηση ***void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pcTaskName)*** η οποία και περιέχει τον κώδικα που έχει γράψει ο χρήστης για να γίνει handle το overflow. Η API function για την δημιουργία ενός task είναι η εξής:
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, uint16_t usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask). Η πρώτη παράμετρος είναι ένας pointer στη C συνάρτηση με τον κώδικα που θα τρέχει το task, η δεύτερη είναι ένα όνομα για το task απλά και μόνο για σκοπούς debugging, η τρίτη το μέγεθος της μνήμης που θα εκχωρηθεί ως στοίβα usStackDepth*stack_width bytes, η τέταρτη για να δοθούν, προαιρετικά, παράμετροι στην function του task, η πέμπτη είναι πολύ βασική έννοια του FreeRTOS κι είναι η προτεραιότητα που δίνεται αρχικά στο task και τέλος ένας pointer σαν handler στο task που θα δημιουργηθεί (ώστε να μπορεί μια άλλη task που έχει τον handler να μπορεί,για παράδειγμα, να αλλάξει την προτεραιότητα του ή να το διαγράψει).

Το priority του task δείχνει ,στην συνηθισμένη περίπτωση που έχουμε αρκετά task, ποιο θα επιλεχθεί να τρέξει αν παραπάνω από ένα είναι έτοιμο προς εκτέλεση.Η προτεραιότητα είναι μεν μια τιμή που δίνεται στην δημιουργία του task αλλά μπορεί να αλλάξει σε οποιοδήποτε στιγμή της ζωής του. Ένα συνηθισμένο παράδειγμα εφαρμογής, είναι να υπάρχει ένα task χαμηλής προτεραιότητας σαν οδηγός οθόνης και ένα άλλο task μεγαλύτερης προτεραιότητας το οποίο ανταποκρίνεται στο πάτημα ενός κουμπιού. Επομένως tasks τα οποία είναι κρίσιμο να τρέξουν άμεσα ,μόλις γίνουν έτοιμα για εκτέλεση,θα πρέπει να τίθενται σε μεγαλύτερη προτεραιότητα.

2.2.2 Χρονοδιακοπές FreeRTOS Tick

Στην συνηθισμένη λειτουργία του το FreeRTOS βασίζεται σε περιοδικά interrupt τα λεγόμενα “ticks” που συμβαίνουν με περίοδο που τίθεται στο αρχείο *FreeRTOSConfig.h*. Λέμε συνηθισμένη διότι πλέον υποστηρίζεται και tickless mode σε περιπτώσεις που το ζητούμενο είναι η ελαχιστοποίηση της κατανάλωσης ισχύος,ωστόσο κάτι τέτοιο δεν θα μας απασχολήσει στα πλαίσια της παρούσας εργασίας. Μια τυπική τιμή είναι 100 Hz(περίοδος 10ms) κι ανάλογα με τον επεξεργαστή στον οποίο έχει γίνει port το FreeRTOS επιλέγεται και η πηγή αυτών των interrupt.Συγκεκριμένα στο Zynq-700 SoC της Xilinx όπως φαίνεται από το αρχείο *FreeRTOS_tick_config.h* και το αρχείο *xscutimer.h* της Xilinx, πηγή είναι ο Private Timer του ARM core. Σε κάθε tick interrupt τρέχει η ***FreeRTOS_Tick_Handler()*** που χονδρικά εκτελεί τον παρακάτω ψευδοκώδικα,σχήμα 2.2.2.1.



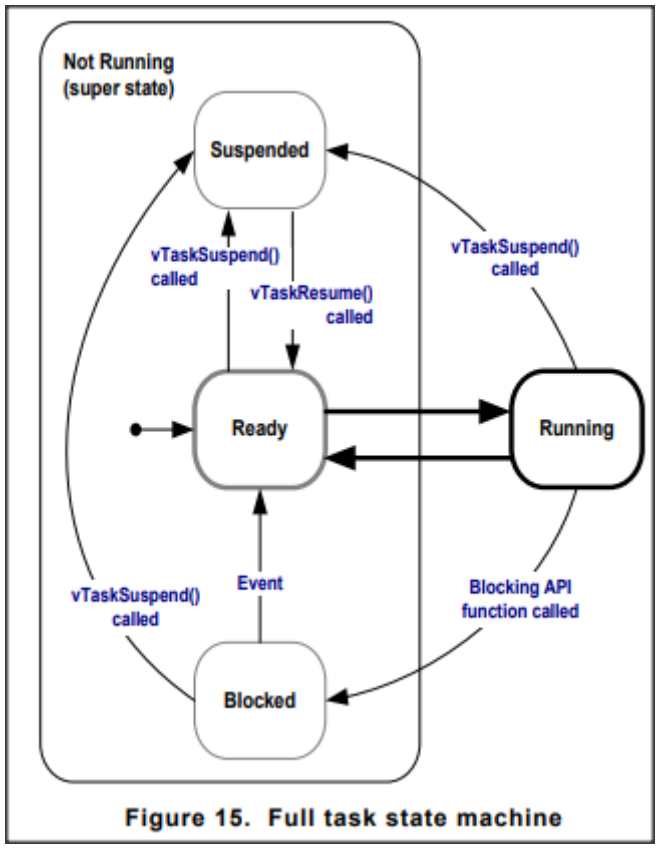
Σχήμα 2.2.2.1 :Ψευδοκώδικας της ρουτίνας του FreeRTOS για εξυπηρέτηση του tick interrupt[6].

Όπως βλέπουμε δηλαδή υπάρχει η πιθανότητα να γίνει context switch μόλις συμβεί ένα tick interrupt και έπειτα να αρχίσει να τρέχει ένα άλλο task.

Γενικά η περίοδος των tick αποτελεί την μονάδα χρόνου για το FreeRTOS με την έννοια ότι τα system calls του, όταν περιέχουν κάποια παράμετρο χρόνου, αναμένουν τα χρονικά διαστήματα σαν πολλαπλάσια αυτής της περιόδου δηλαδή σε αριθμό ticks.Για παράδειγμα αν θέλουμε ένα task να κάνει delay 5ms γράφουμε ***vTaskDelay(pdMS_TO_TICKS(5))*** οπότε και γίνεται μετατροπή σε “ticks” τα οποία κι αναμένει η ***vTaskDelay***. Η παραπάνω κλήση θέτει το task σε blocked state.

2.2.3 Οι καταστάσεις των διεργασιών του FreeRTOS

Παραπάνω αναφέρθηκε ότι η κλήση στην ***vTaskDelay*** θέτει την διεργασία σε κατάσταση blocked για όσο χρονικό διάστημα που επιθυμούμε αναμονή. Γενικά το σύνολο των καταστάσεων(states) που μπορεί να βρίσκεται ένα FreeRTOS task είναι: Ready, Running, Blocked, Suspended όπως φαίνεται στο παρακάτω διάγραμμα καταστάσεων. Επιπλέον φαίνονται κι οι πιθανές μεταβάσεις στο σχήμα 2.2.3.1.



Σχήμα 2.2.3.1 :Διάγραμμα καταστάσεων των FreeRTOS tasks[6].

Η μετάβαση από Running σε Ready γίνεται για παράδειγμα στην περίπτωση που περισσότερες της μιας διεργασίες ίσης προτεραιότητας είναι Ready οπότε και γίνεται context switch σε κάθε tick.Ή στην περίπτωση που κάποια διεργασία ανώτερης προτεραιότητας έγινε Ready οπότε κι άρχισε να εκτελείται,μετά από context switch της διεργασίας μικρότερης προταιρετότητας.

2.2.4 Χρονοπρογραμματισμός των διεργασιών FreeRTOS scheduling

Το πρώτο από τα παραπάνω παραδείγματα αποτελεί περίπτωση Round Robin χρονοδρομολόγησης στην οποία έχουμε μεταβάσεις από Running σε Ready και το αντίστροφο κάθε φορά που συμβαίνει tick interrupt. Αυτό συμβαίνει στην συνηθισμένη περίπτωση στην οποία στο αρχείο FreeRTOSConfig.h έχει επιλεγεί Fixed Priority Pre-emptive Scheduling with Time Slicing με τις παρακάτω 2 επιλογές ***configUSE_PREEMPTION 1*** και ***configUSE_TIME_SLICING 1***.Η επιλογή ***configUSE_PREEMPTION*** σημαίνει ότι ο ένα task μπορεί, μη οικειοθελώς, να μεταβεί από Running σε Ready state ώστε να τρέξει κάποιο άλλο task. Με αυτές τις επιλογές έχουμε

context switch στις ακόλουθες περιπτώσεις: όταν ένα υψηλότερης προτεραιότητας task μπει σε Ready state, όταν το task που βρίσκεται σε εκτέλεση μπει σε Blocked ή Suspended State κι όταν, έχοντας επιλέξει time sharing, συμβεί tick interrupt. Η δεύτερη επιλογή **configUSE_TIME_SLICING**, όπως δηλώνει το όνομα της, είναι υπεύθυνη για το διαμοιρασμό του χρόνου επεξεργαστή μεταξύ ίσης προτεραιότητας task και context switch σε κάθε tick. Αν η **configUSE_TIME_SLICING** τεθεί στο 0 τότε δεν έχουμε συμπεριφορά processor sharing. Τώρα στην περίπτωση που η **configUSE_PREEMPTION** τεθεί στο 0 λέμε ότι χρησιμοποιείται co-operative scheduler και τότε ένα Running task παραχωρεί την εκτέλεση σε ένα άλλο είτε όταν μπει σε Blocked state, είτε όταν καλέσει την εντολή ***taskYIELD()***. Προφανώς στην περίπτωση αυτή δεν έχει νόημα η επιλογή **configUSE_TIME_SLICING** αφού ποτέ ένα task δεν μπορεί να διακοπεί χωρίς την ‘’θέληση” του, επομένως ούτε σε ένα tick interrupt.

Το ίδιο το FREE_RTOS δημιουργεί τα εξής δύο task: idle task και daemon task. Το idle task έχει την ελάχιστη προτεραιότητα (PRIORITY 0) και κατά συνέπεια τρέχει όταν άλλα ανώτερης προτεραιότητας task είναι Blocked ενώ μπορεί να μοιράζεται χρόνο επεξεργαστή με άλλα task ίσης προτεραιότητας τα οποία έχουν οριστεί από τον σχεδιαστή του συστήματος. Είναι, όμως, απαραίτητο να λαμβάνει σποραδικά κάποιο χρόνο επεξεργαστή διότι εκεί βρίσκεται ο κώδικας που απελευθερώνει την μνήμη μετά την διαγραφή ενός task.Το daemon task,όπως ονομάζεται στην ορολογία του FreeRTOS, έχει προτεραιότητα οριζόμενης στο αρχείο FreeRTOSConfig.h και, εκτός άλλων, περιέχει τον κώδικα για την υποστήριξη των λεγόμενων software timer που θα δούμε παρακάτω.

2.2.5 Ανταλλαγή δεδομένων μεταξύ διεργασιών

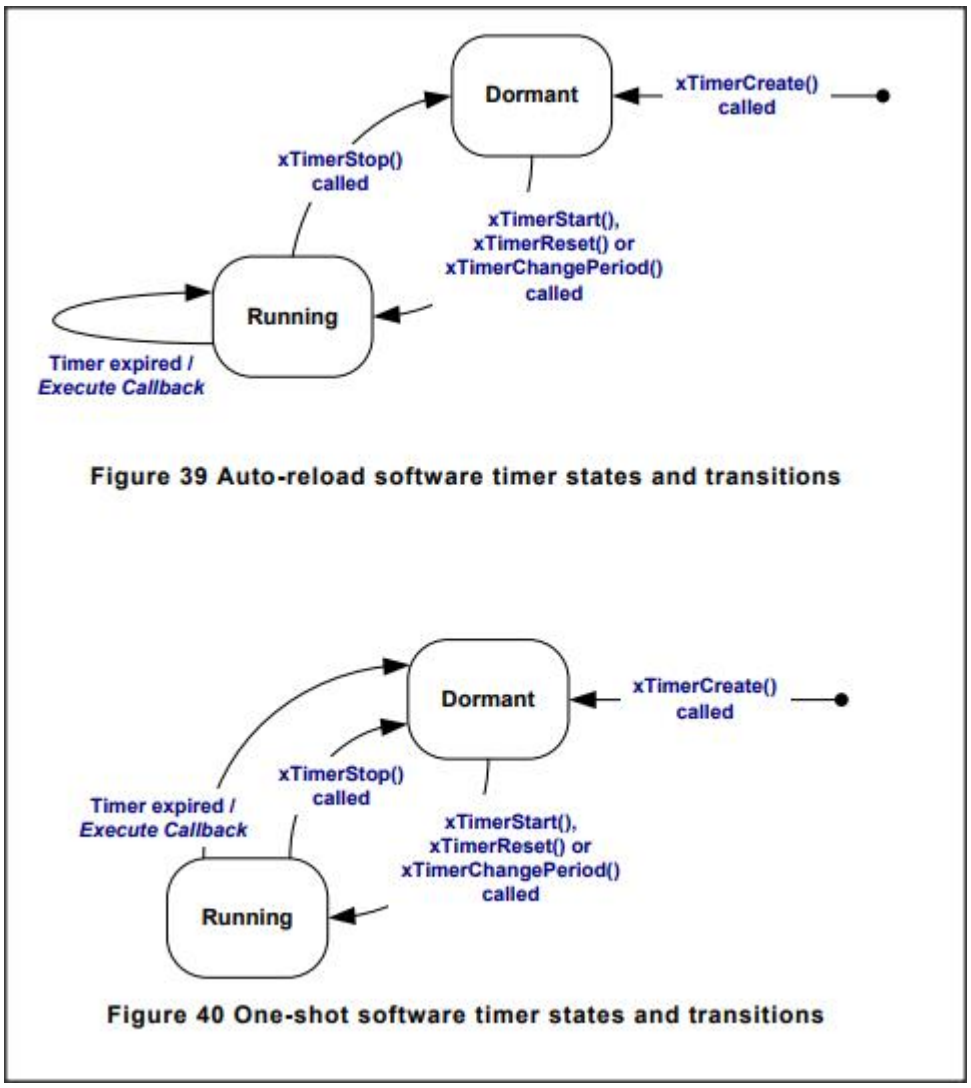
Η δυνατότητα ανταλλαγής δεδομένων μεταξύ των task στο FreeRTOS υποστηρίζεται με τα λεγόμενα queues. Πρόκειται για μια έννοια παρόμοια με FIFO δομή στην οποία τα task εγγραφείς γράφουν στο τέλος της και τα tasks αναγνώστες διαβάζουν από την αρχή. Κλήσεις που αφορούν ένα Queue read ή write μπορεί να οδηγήσουν το καλόν task σε Blocked state αν η ουρά είναι άδεια η γεμάτη αντίστοιχα.

Για την χρήση του μηχανισμού, αρχικά θα πρέπει να δημιουργηθεί το queue με μια κλήση στην ***QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize)*** η οποία αν είναι επιτυχής επιστρέφει έναν handler στο queue που δημιουργήθηκε. Στην συνέχεια κάθε task το οποίο διαθέτει το handler στο queue μπορεί να καλέσει τις **BaseType_t xQueueSendToFront(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait)** ή
BaseType_t xQueueSendToBack(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait) ώστε να βάλει δεδομένα στην αρχή ή στο τέλος της ουράς (υπάρχει η δυνατότητα για τοποθέτηση και στην αρχή της ουράς) και την **BaseType_t xQueueReceive(QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait)** ώστε να λάβει δεδομένα από την ουρά.

2.2.6 Software Timers

Οι software timers είναι οντότητες του FreeRTOS που έχουν σκοπό να εξασφαλίσουν την εκτέλεση ενός C function σε αυστηρά καθορισμένες χρονικές στιγμές είτε μια φορά, one-shot timer, είτε περιοδικά, auto-reload timer. Για παράδειγμα μπορούμε να πούμε ότι θέλουμε μια συνάρτηση να εκτελεστεί σε 7 ticks όπως συνάρτηση **BacklightOff()** που είναι υπεύθυνη για το σβήσιμο του φωτισμού μιας οθόνης.Μπορεί όμως να επιθυμούμε την εκτέλεση κάποιας ενέργειας αυστηρά κάθε 6 ticks όπως την εκτέλεση της συνάρτηση **LedToggle()** που

αναβοσβήνει ένα ενδεικτικό Led. Μετά την δημιουργία καθενός timer αυτός μπορεί να βρίσκεται είτε σε running είτε σε dormant state όπως φαίνεται στο σχήμα 2.2.6.1.



Σχήμα 2.2.6.1 :Διάγραμμα καταστάσεων των FreeRTOS software timer[6].

Όπως αναφέρθηκε πριν η daemon task είναι υπεύθυνη για την διαχείριση των software timer. Όταν τα task θέλουν να τους χρησιμοποιήσουν καλούν τις API συναρτήσεις που παρέχει το FreeRTOS. Αυτές γράφουν τις εντολές (create, start, reset, change_period, stop) μαζί με ένα timestamp σε ένα queue που ονομάζεται command queue.Το daemon task διαβάζει από το command queue και είναι Blocked όσο το queue είναι άδειο. Αν υποθέσουμε ότι το daemon task έχει αρκετά μεγάλη προτεραιότητα,όταν δοθεί ένα command από κάποιο άλλο task τότε αμέσως θα γίνει context switch και θα αρχίσει να εκτελείται το daemon task, το οποίο θα γράψει στις δομές δεδομένων που αφορούν τους software timers. Αν δεν έχουν φτάσει άλλα δεδομένα-εντολές στην command queue το daemon task επιστρέψει σε Blocked state. Επίσης θα πρέπει να αναφερθεί ότι ο χρόνος που θέτουμε σε ένα software timer call, του πότε δηλαδή θέλουμε να τρέξει η C συνάρτηση που ελέγχεται από τον software timer, δεν επηρεάζεται από το πότε θα τρέξει το daemon task επειδή κάθε εντολή που μπαίνει στο command queue έχει ένα timestamp (σε ticks counter). Προφανώς αν το daemon δεν τρέξει σε χρονικό διάστημα μικρότερο από το ζητούμενο ο χρονισμός χάνεται κι είναι ευθύνη του προγραμματιστή η ανάθεση προτεραιοτήτων στο daemon όπως και στα υπόλοιπα tasks.

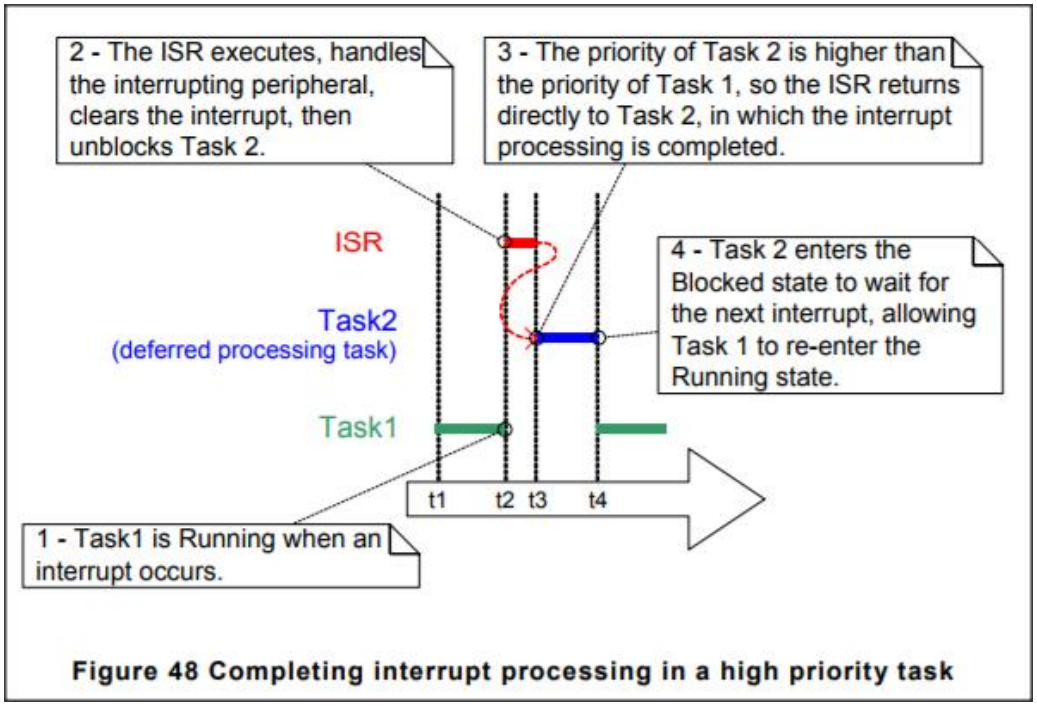
2.2.7 Διακοπές και FreeRTOS

Οι διακοπές (interrupts) αποτελούν πολύ χρήσιμο μηχανισμό για τα ενσωματωμένα συστήματα διότι τα κάνουν οδηγούμενα από γεγονότα (event driven) με αποτέλεσμα τον περιορισμό της χρήσης του polling το οποίο οδηγεί σε σπατάλη επεξεργαστικών πόρων κι ενέργειας. Η χρήση του FreeRTOS δεν παρεμποδίζει σε καμία περίπτωση την χρήση μηχανισμών διακοπών, μάλιστα βασίζεται σε αυτά όπως είδαμε τα tick interrupts, και μάλιστα παρέχει ένα σύνολο API συναρτήσεων οι οποίες είναι interrupt safe κι επομένως μπορούν να κληθούν μέσα από το σώμα μιας ISR (Interrupt Service Routine). Θα αναφερθεί ένα παράδειγμα μιας τυπικής εφαρμογής: έστω πως θέλουμε να ενεργοποιείται ο φωτισμός μιας οθόνης κάθε φορά που πιέζουμε ένα πλήκτρο και να μένει ενεργοποιημένος για 10 ticks μετά,αν, επιπλέον, πιεστεί ξανά το πλήκτρο ο χρόνος θα πρέπει να ανανεώνεται. Μπορούμε λοιπόν να αναθέσουμε την συνάρτηση **BacklightOff()** σε ένα one-shot software timer με χρόνο 10 ticks και μέσα στην ISR του πλήκτρου να ενεργοποιούμε τον οπίσθιο φωτισμό και να καλούμε την συνάρτηση **xTimerResetFromISR(...)** για τον παραπάνω software timer. Συνεπώς κάθε φορά που πιέζουμε το πλήκτρο, το Backlight ανάβει και δίνουμε χρονικό διάστημα 10 ticks μέχρι να σβήσει.

Η **xTimerResetFromISR(...)** είναι ένα παράδειγμα του interrupt safe API του οποίου η ύπαρξη αποτελεί σχεδιαστική επιλογή του FreeRTOS. Αυτό διότι θα μπορούσε κάθε API call να είναι interrupt safe κι έτσι να μην απασχολούσε τον προγραμματιστή ποιο function θα χρησιμοποιήσει. Ωστόσο μια τέτοια επιλογή θα πρόσθετε overhead σε κάθε API call ακόμα κι όταν αυτό δεν θα χρειάζονταν, με αποτέλεσμα ανεπιθύμητο φορτίο για το σύστημα.
Γενικά υπάρχει η δυνατότητα και για context switch μέσα από interrupt με αποτέλεσμα η ISR να επιστρέψει σε διαφορετικό σημείο κώδικα(!), κάποιου άλλου task. Βέβαια αυτή η λειτουργικότητα εξαρτάται από την αρχιτεκτονική που χρησιμοποιείται και για κάθε port υπάρχει port-specific κώδικας.

2.2.8 Συγχρονισμός

Ο κώδικας μέσα σε μια ρουτίνα εξυπηρηρέτησης διακοπής (ISR) θα πρέπει να είναι όσο το δυνατόν συντομότερος για διάφορους λόγους. Καταρχάς, όταν εκτελείται μια ISR δεν εκτελείται κάποιο task όσο υψηλή προτεραιότητα κι αν έχει, οπότε όσο μεγαλύτερη είναι η ρουτίνα εξυπηρέτησης τόσο μεγαλύτερη καθυστέρηση εισάγεται. Αυτό σε συνδυασμό με την τυχαιότητα των interrupt events εισάγει αβεβαιότητα στην απόκριση του συστήματος, εισάγει το λεγόμενο jitter. Επιπλέον, κατά την διάρκεια εκτέλεσης μια ρουτίνας εξυπηρέτησης έχουμε 2 πιθανές συμπεριφορές στην περίπτωση νέου interrupt ,ανάλογα με την αρχιτεκτονική. Είτε δεν γίνεται εξυπηρέτηση του νέου interrupt μέχρι να τελείωσει η εξυπηρέτηση του πρώτου είτε αν υποστήριζεται interrupt nesting και interrupt priorities είναι πιθανό να γίνει nesting. Στην πρώτη περίπτωση υπάρχει η πιθανότητα να χαθεί κάποιο event ενώ στην δεύτερη αυξάνεται η σχεδιαστική πολυπλοκότητα και το σύστημα γίνεται λιγότερο προβλέψιμο. Άρα είναι κοινή παραδοχή πως η ρουτίνες εξυπηρέτησης διακοπών θα πρέπει να είναι όσο το δυνατόν συντομότερες. Ωστόσο,πολλές φορές ένα event μπορεί να απαιτεί αρκετό χρόνο επεξεργασίας (πχ η λήψη ενός network packet από τον Wifi Controller απαιτεί εκτέλεση TCP/IP κώδικα). Μια λύση σε αυτό το πρόβλημα είναι το interrupt απλά να σηματοδοτεί το γεγονός και να ειδοποιεί το task που θα αναλάβει την επεξεργασία του event πχ TCP/IP task. Κάτι τέτοιο φαίνεται στο σχήμα 2.2.8.1.



Σχήμα 2.2.8.1 :Ολοκλήρωση επεξεργασίας διακοπής σε task υψηλής προταιρεότητας[6].

Κάτι τέτοιο όμως απαιτεί έναν μηχανισμό συγχρονισμού μεταξύ interrupt και task (I2T) όπως binary και counting semaphores,event groups και task notifications όπως ορίζονται στο documentaion του FreeRTOS.

2.3 Σύγκριση FreeRTOS και Linux

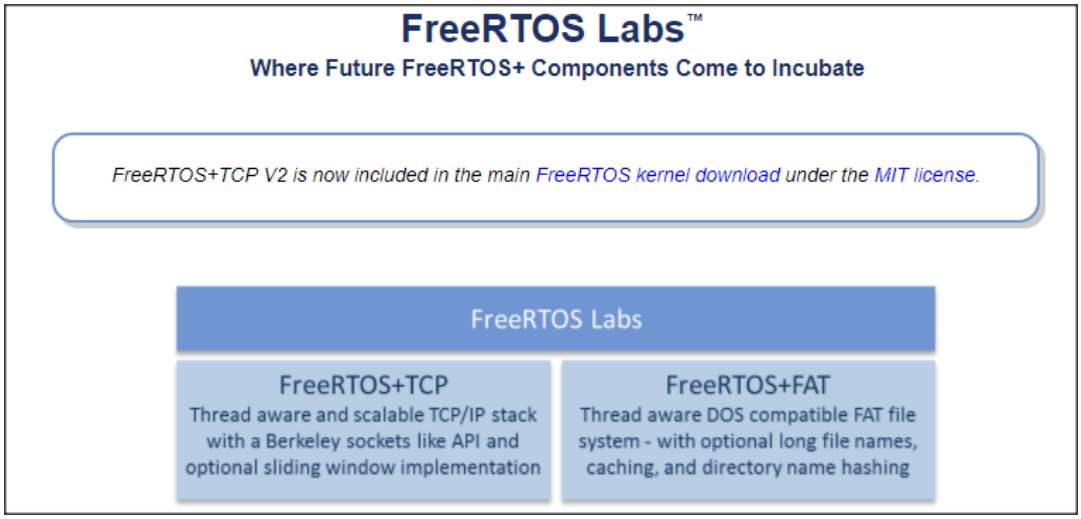
Στο κεφάλαιο αυτό γίνεται μια σύγκριση του λειτουργικού συστήματος γενικού σκοπού (General Purpose Operating System,GPOS) Linux και του λειτουργικού πραγματικού χρόνου RTOS (Real Time Operating System) FreeRTOS. Τα σημεία της σύγκρισης ακολουθούν παρακάτω.

Scheduling και λειτουργίες

Στο FreeRTOS μπορούν να δοθούν εγγυήσεις για ντετερμινιστική συμπεριφορά του συστήματος με αυστηρά χρονικά όρια εκτέλεσης λειτουργίων κι εξυπηρέτησης γεγονότων (μπορεί να είναι και περιοδικά όπως ένας βρόχος PID ελέγχου). Ο βασικός μηχανισμός,όπως έχουμε δει, είναι η preemptive δρομολόγηση με προτεραιότητες στα tasks. Το task με την μεγαλύτερη προτεραιότητα θα τρέξει μόλις γίνει ready διακόπτοντας την εκτέλεση άλλων task. Αυτό μπορεί να προκληθεί, για παράδειγμα, από ένα interrupt, την εγγραφή δεδομένων σε μια ουρά ή με κάποιο μηχανισμό συγχρονισμού. Το Linux, αν και παρέχει επιλογές δρομολόγησης ακόμη κι έναν real time scheduler δεν μπορεί να παρέχει αυστηρές (hard) real time εγγυήσεις διότι η μεγαλύτερη πολυπλοκότητα του κώδικα πυρήνα, σε σχέση με το FreeRTOS, εισάγει πολύ μεγαλύτερες καθυστερήσεις (latencies), ακόμα και 100 φορές πιο πάνω για ένα απλό context switch σε σύγκριση με το FreeRTOS.

Δυνατότητες

Εδώ εξετάζουμε ποιες λειτουργίες παρέχει έτοιμες στον προγραμματιστή η χρήση του κάθε λειτουργικού συστήματος. Το Linux εδώ υπερτερεί καθώς προσφέρει όλες τις γνωστές δυνατότητες ενός σύγχρονου λειτουργικού όπως ένα γραφικό περιβάλλον, TCP/IP stack, σύστημα αρχείων, C και Python compiler καθώς κι οποιοδήποτε package κώδικα από την κοινότητα. Το FreeRTOS από την άλλη, προσφέρει τα βασικά, κι απαραίτητα, που θα πρέπει να προσφέρει ένα λειτουργικό, δηλαδή τον scheduler, μηχανισμούς επικοινωνίας μεταξύ των tasks(IPC interprocess communication) και μηχανισμούς συγχρονισμού. Πέρα από αυτά οποιαδήποτε παραπάνω λειτουργικότητα μπορεί να ενσωματωθεί αλλά είναι ευθύνη του σχεδιαστή να προσθέσει τον (πηγαίο) κώδικα. Κώδικας που είτε θα αναπτυχθεί από την ομάδα που σχεδιάζει το σύστημα (in-house) είτε από τρίτους (third party).Η ομάδα του FreeRTOS αναπτύσσει επεκτάσεις όπως TCP/IP stack και FAT σύστημα αρχείων,σχήμα 2.3.1:

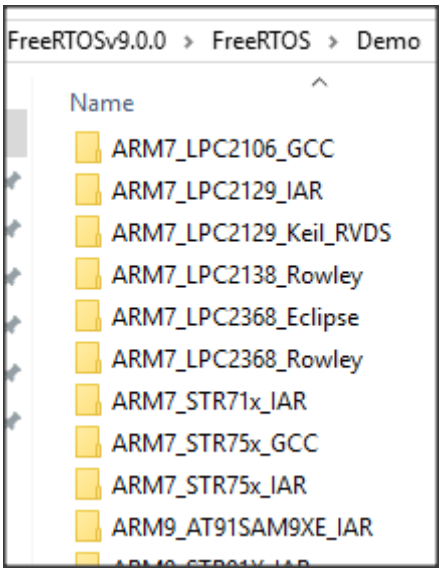


Σχήμα 2.3.1 :FreeRTOS TCP/IP stack, FreeRTOS FAT file system.

Η επιλογή τους έχει σημαντικά πλεονεκτήματα με κυριότερα την άριστη συνεργασία με τον πυρήνα του FreeRTOS και το υψηλό επίπεδο ποιότητας στο οποίο κινείται γενικότερα το FreeRTOS project.

Ευκολία ανάπτυξης

Τόσο το Linux όσο και το FreeRTOS προσφέρουν εκτενέστατο documentation για τις λειτουργίες τους. Κάτι τέτοιο είναι γνωστό για το Linux αλλά όχι για το FreeRTOS. Πιο συγκεκριμένα το αρχείο *"Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide"* [6] περιέχει μια πλήρη περιγραφή των λειτουργιών με παραδείγματα για την κάθε περίπτωση ελαχιστοποιώντας έτσι την περίοδο εκμάθησης. Επίσης, στο φάκελο λήψης του FreeRTOS υπάρχει ο υποφάκελος Demos,σχήμα 2.3.2.



Σχήμα 2.3.2 :Τα demo για κάθε port του FreeRTOS.

Ο οποίος περιέχει έτοιμα παραδείγματα για το κάθε target. Αυτά χρησιμεύουν για την γρήγορη επιβεβαίωση σωστής λειτουργίας κι επιβεβαίωση της αναμενόμενης συμπεριφοράς σε ένα νέο σύστημα και συνιστώνται ως σημείο αφετηρίας για την ανάπτυξη της επιθυμητής εφαρμογής με διαδοχικές τροποποιήσεις.

Τρόπος λειτουργίας και μια βασική διαφορά

Στο σημείο αυτό θα πρέπει να αναφερθεί μια πολύ σημαντική διαφορά μεταξύ FreeRTOS και Linux. Όπως είναι γνωστό, σε περιβάλλον Linux μπορούμε να φορτώσουμε και να τρέξουμε δυναμικά εφαρμογές οι οποίες βρίσκονται σε κάποιο μέσο αποθήκευσης. Το εκτελέσιμο του πυρήνα είναι ανεξάρτητο από τον κώδικα οποιασδήποτε εφαρμογής. Το FreerRTOS, από την άλλη, είναι στην ουσία μια C βιβλιοθήκη και σε συνδυασμό με τον κώδικα της εφαρμογής, δηλαδή των κώδικα των task, παράγεται ένα ενιαίο compilation image για το εκάστοτε target. Η μεταγλώττιση της εφαρμογής δεν είναι ανεξάρτητη από την μεταγλώττιση του πυρήνα, και δεν υποστηρίζεται δυναμική φόρτωση κώδικα από ένα σύστημα αρχείων όπως στο Linux.

Εικονική μνήμη

Μια ακόμα βασική διαφορά είναι ότι στο FreeRTOS δεν έχουμε μηχανισμό μετάφρασης διευθύνσεων δηλαδή εικονική μνήμη,σε αντίθεση με το Linux. Οπότε στο FreeRTOS κάθε task “βλέπει” φυσικές διευθύνσεις χωρίς περιορισμό και συνεπώς κώδικας με σφάλματα σε ένα task μπορεί να προκαλέσει αλλοίωση δεδομένων και αστάθεια σε όλο το σύστημα. Όταν όμως χρησιμοποιείται εικονική μνήμη εισάγεται αβεβαιότητα στον χρονισμό κατά το paging, δηλαδή δεν είναι εκ των προτέρων γνωστή η διάρκεια των προσβάσεων στη μνήμη λόγω πιθανών λειτουργιών οργάνωσης. Θα πρέπει να σημειωθεί ότι για τα targets ARM Cortex-M3 και ARM-Cortex-M4F υπάρχουν δύο ports του FreeRTOS, το βασικό port κι ένα που εκμεταλλεύεται την μονάδα MPU(Memory Protection Unit) την οποία ενσωματώνουν αυτές οι δύο αρχιτεκτονικές. Οι βασικές λειτουργίες του FreeRTOS-MPU είναι ότι μπορεί κάθε task να έχει διαφορετικά δικαιώματα σε διαφορετικές περιοχές μνήμης ενώ πλέον γίνεται διαχωρισμός σε Privilleged και User tasks, όπως ονομάζονται,ανάλογα με το επίπεδο δικαιωμάτων. Μπορεί μεν να μην υπάρχουν εικονικές διευθύνσεις και μετάφραση αλλά δίνονται λύσεις σε σημαντικά προβλήματα,όπως εγγυημένη ανίχνευση stack overflow κι έτσι το σύστημα γίνεται περισσότερο εύρωστο (robust) κι αξιόπιστο.

Μέγεθος

Η διαφορά σε απαιτήσεις μνήμης μεταξύ Linux και FreeRTOS είναι πολύ μεγάλες τόσο σε μη πτητική μνήμη (μνήμη εντολών και αρχικοποιημένων δεδομένων) όσο και σε μνήμη RAM (στοίβα και σωρός). Ένα χαρακτηριστικό παράδειγμα για το πόσο μικρές απαιτήσεις μνήμης μπορεί να έχει ένα port του FreeRTOS είναι στην περίπτωση του AVR. Σε 8-bit AVR το αποτύπωμα (footprint) στην FLASH μνήμη είναι της τάξης των 5 kilobytes, ενώ με την υπόθεση ότι στο σύστημα έχουμε 2 task,μια ουρά(queue) για επικοινωνία καθώς κι έναν σημαφόρο συγχρονισμού απαιτούνται μόλις 200 bytes RAM, χωρίς βέβαια να υπολογίζουμε τον κώδικα των task.

Στο Linux από την άλλη υπάρχει η δυνατότητα να αφαιρεθούν τμήματα του πυρήνα για λειτουργίες που δεν θα χρησιμοποιηθούν κι έτσι να μειωθεί το αποτύπωμα μνήμης αλλά και πάλι η τάξη μεγέθους είναι megabyte.

Επεξεργαστική ισχύς

Η απλότητα και το μικρό μέγεθος του κώδικα του FreeRTOS σημαίνουν και μικρό overhead στην λειτουργία του συστήματος για τις διάφορες λειτουργίες όπως ένα context switch. Επιπλέον συγκρίνοντας πάνω στην ίδια αρχιτεκτονική το Linux εισάγει ένα σημαντικό boot time που θα πρέπει να ληφθεί υπόψη από τον σχεδιαστή του συστήματος.

Targets

Οι δύο παραπάνω παράγοντες δηλαδή η μνήμη και η επεξεργαστική ισχύς καθορίζουν το εύρος των διαθέσιμων αρχιτεκτονικών στις οποίες μπορεί να εγκατασταθεί το κάθε λειτουργικό σύστημα. Για το FreeRTOS μάλιστα η δυνατότητα του porting σε όσους περισσότερους (μίκρο)επεξεργαστές αποτέλεσε κι αποτελεί στόχο για την ομάδα ανάπτυξης έτσι ώστε να σήμερα να υποστηρίζονται περίπου 160,με εύρος από 8-bit AVR εώς 32-bit

ARM Cortex-A. Πολλά ports του FreeRTOS εκμεταλλεύονται τις extra δυνατότητες της υποκείμενης αρχιτεκτονικής όπως για παράδειγμα FPU(Floating-point Unit) και MPU(Memory Protection Unit), όπως είδαμε πριν. Το (embedded) Linux, από την άλλη, πέρα από τους γνωστούς x86, όπως το Intel Quark SoC X1000 στο Intel Galileo στο πεδίο των ενσωματωμένων συστημάτων, κατά πλειοψηφία σήμερα συναντάται να τρέχει στην high-end γκάμα των ARM επεξεργαστών. Βέβαια κι η ομάδα του Linux προσφέρει παραλλαγές όπως για παράδειγμα στην περίπτωση που δεν υπάρχει MMU(Memory Manageme Unit) αλλά υπάρχει MPU(Memory Protection Unit) προσφέρεται η τροποποιημένη έκδοση uClinux.

Επίλογος

Όπως ήδη θα έχει γίνει κατανοητό η σύγκριση δεν είναι απόλυτα ευσταθής διότι η επιλογή τελικώς ανάγεται στο αν ο σχεδιαστής επιθυμεί την απλότητα, το μικρόμέγεθος και την ταχύτητα του FreeRTOS ή το πλήρες εύρος δυνατοτήτων ενός σύγχρονου λειτουργικού όπως το Linux. Αν για παράδειγμα το σύστημα που σχεδιάζεται διαβάζει δεδομένα από αισθητήρες, ελέγχει έναν ηλεκτροκινητήρα και παράλληλα έχει μια οθόνη με απλά γραφικά για παραμετροποίηση και monitoring, τότε σίγουρα η επιλογή του FreeRTOS είναι η κατάλληλη. Κι αυτό διότι η απλότητα του λογισμικού θα συντομεύσει το TTM(Time to Market) κι επίσης υπάρχει ευελιξία στην επιλογή μικροεπεξεργαστή που μπορεί να οδηγήσει σε χαμηλότερο κόστος. Από την άλλη,αν το υπό σχεδιασμό σύστημα χρειάζεται να διαβάζει αισθητήρες,να στέλνει email με αναφορές μετρήσεων και να τρέχει έναν Web server και Node.js, η καλύτερη επιλογή θα ήταν να χρησιμοποιηθεί Linux. Κι αυτό γιατί υπάρχει έτοιμος κώδικας που έχει δοκιμασθεί για τις παραπάνω λειτουργίες τον οποίο διαφορετικά θα έπρεπε να ξαναγράψει η ομάδα ανάπτυξης. Επίσης εφόσον οι απαιτήσεις είναι προχωρημένες, ούτως ή άλλως θα επιλεχθεί ένας high-end επεξεργαστής κι άρα δεν λαμβάνεται υπόψη αυτή η παράμετρος.

Τέλος,το FreeRTOS δίνει την δυνατότητα της χρήσης του σε εμπορικά προϊόντα και με τροποποίηση του κώδικα του πυρήνα χωρίς να πρέπει να δημοσιευθεί ο κώδικας της εφαρμογής, σε αντίθεση με την άδεια που παρέχει το Linux. Εδώ, ο οικονομικός παράγοντας υπεισέρχεται άμεσα αφού στην δεύτερη περίπτωση θα πρέπει η εταιρία που διαθέτει το προϊόν να δαπανεί για άδειες χρήσης. Άρα καταλήγουμε στο γνωστό συμπέρασμα πως θα πρέπει να χρησιμοποιείται το κατάλληλο εργαλείο για την κατάλληλη εφαρμογή και πως ο σχεδιαστής θα πρέπει να γνωρίζει και τα δύο για να κάνει την καταλληλότερη επιλογή.

Συνοπτικά:

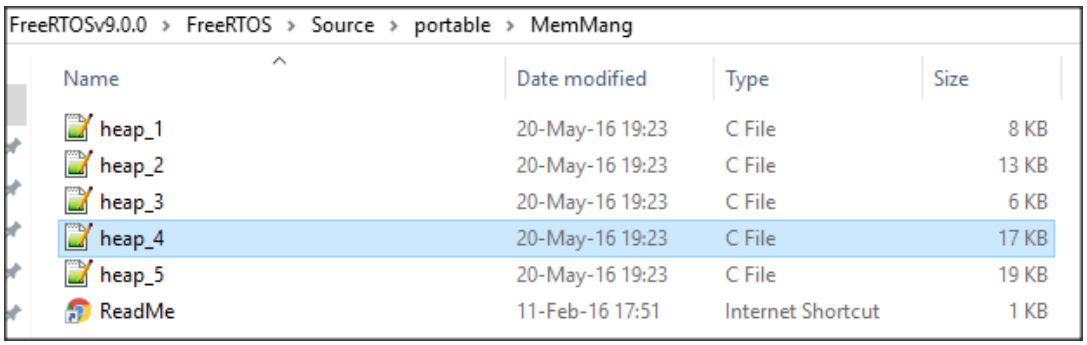
Παράμετρος	**FreeRTOS**	**Linux**
Διαχείριση διεργασιών(Scheduling)	Δυνατότητα hard-real time χρονικών ορίων	Περιορισμοί στα χρονικά όρια που μπορούν να τεθούν
Δυνατότητες	Παρέχονται έτοιμες μόνο οι βασικές λειτουργίες ενός λειτουργικού συστήματος	Τεράστια γκάμα λειτουργιών όπως γραφικό περιβάλλον,C compiler,σύστημα αρχείων κλπ.
Ευκολία ανάπτυξης	Πολύ απλό και περιεκτικό documentation και demo για κάθε port	Εκτενέστατο documentation και support από την κοινότητα

Δυναμική φόρτωση διεργασιών	Δεν υποστηρίζεται	Φόρτωση εφαρμογών από σύστημα αρχείων
Εικονική μνήμη	Δεν υλοποιείται εν γένει, μόνο σε ορισμένα targets έχουμε δυνατότητα προστασίας μνήμης	Πλήρης υποστήριξη
Μέγεθος	Τάξης μερικών KB στα μικρά targets	Τάξης μερικών MB
Απαιτούμενη επεξεργαστική ισχύς	Μικρός κι απλός κώδικας πυρήνα,μικρές απαιτήσεις	Σημαντικές απαιτήσεις σε επεξεργαστική ισχύ ώστε να μειωθεί το αντίκτυπο των διάφορων overhead
Διαθέσιμες αρχιτεκτονικές (Targets)	Περίπου 160 ports, από 8bit AVR ως 32-bit ARM Cortex-A	Κατά βάση x86,x64,amd64 και high-end ARM

Σχήμα 2.3.3 :Σύνοψη της σύγκρισης FreeRTOS και Linux.

2.4 Επιλογές διαχείρισης δυναμικής μνήμης στο FreeRTOS

Η διαχείριση μνήμης είναι μια έννοια που, σε πρώτο επίπεδο, δεν έχει να κάνει με την υλοποίηση του FreeRTOS ή την χρονοδρομολόγηση των task. Ωστόσο ο πυρήνας του FreeRTOS δημιουργεί και διαχειρίζεται αντικείμενα δυναμικά κι έτσι υπάρχει η ανάγκη για δυναμική διαχείριση μνήμης.Στο φάκελο λήψης του FreeRTOS υπάρχει ο παρακάτω υποφάκελος,σχήμα 2.4.1.



Σχήμα 2.4.1 :Οι 5 υλοποιήσεις δυναμικής διαχείρισης μνήμης στο FreeRTOS.

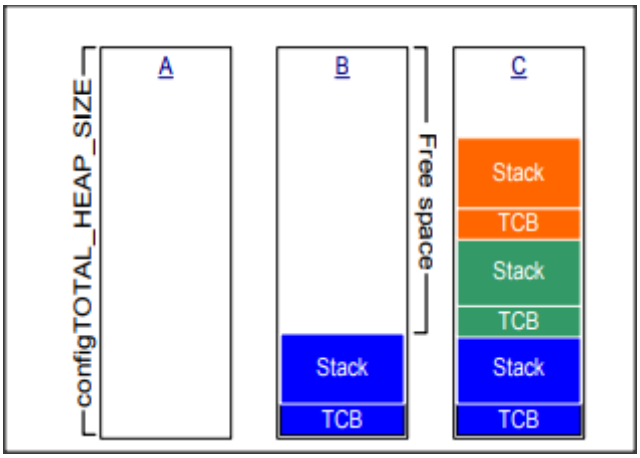
Κάθε ένα αρχείο παρέχει και μια διαφορετική υλοποίηση των συναρτήσεων pvPortMalloc() και vPortFree() οι οποίες καλούνται από άλλα σημεία του κώδικα του FreeRTOS για την δυναμική εκχώρηση μνήμης όπως για παράδειγμα κατά την δημιουργία ενός νέου task όπου εκχωρείται στοίβα καθώς και το TCB (Task Control Block) με τα απαραίτητα δεδομένα για τη διαχείριση του task από το λειτουργικό. Οι λόγοι που δεν χρησιμοποιούνται οι C standard συναρτήσεις malloc() και free() είναι αρκετοί. Καταρχάς,δεν είναι πάντα διαθέσιμες σε περιπτώσεις μικρών συστημάτων ενώ όπου είναι διαθέσιμες η υλοποίηση τους μπορεί να είναι αρκετά μεγάλη και να καταλαμβάνει πολύτιμο χώρο. Επίσης, οι κλήσεις αυτών των

συναρτήσεων δεν είναι ντετερμινιστικές δηλαδή ο χρόνος εκτέλεσης διαφέρει από κλήση σε κλήση,ανάλογα την κατάσταση της μνήμης όπως για παράδειγμα αν υπάρχει κατακερματισμός σε μεγάλο βαθμό και γενικότερα είναι αρκετά αργές.

Μέχρι την έκδοση V9.0.0 του FreeRTOS όλα τα αντικείμενα που διαχειρίζεται ο πυρήνας λαμβάνουν χώρο μνήμης δυναμικά δηλαδή σε χρόνο εκτέλεσης(run-time). Κάθε φορά που δημιουργείται ένα αντικείμενο πυρήνα δεσμεύεται μνήμη και κάθε φορά που διαγράφεται ελευθερώνεται. Με αυτόν το τρόπο μειώνεται ο χρόνος σχεδιασμού αφού το σύστημα είναι απλούστερο και περιορίζεται το αποτύπωμα (footprint) στην RAM. Από την έκδοση V9.0.0 δίνεται η δυνατότητα στατικής παραχώρησης μνήμης δηλαδή σε χρόνο μεταγλώττισης (compilation-time).
Στην συνέχεια γίνεται παρουσίαση της κάθε μιας υλοποίησης heap_x.c:

Heap_1
Αυτή η υλοποίηση είναι κατάλληλη για συστήματα τα οποία δημιουργούν όλα τα αντικείμενα πυρήνα (πχ tasks) πριν ξεκινήσει να τρέχει ο scheduler. Συνεπώς δεν μας ενδιαφέρει ο ντετερμινισμός των κλήσεων ή ο κατακερματισμός της μνήμης αλλά άλλα χαρακτηριστικά της υλοποίησης όπως το μικρό μέγεθος κι η απλότητα. Έτσι έχουμε μια πολύ απλή υλοποίηση της pvPortMalloc() ενώ η vPortFree() δεν υλοποιείται κι επομένως η επιλογή αυτή προορίζεται για εφαρμογές που δεν διαγράφουν αντικείμενα πυρήνα. Μάλιστα οι κλήσεις στην pvPortMalloc() διαρκούν το ίδιο κάθε φορά δηλαδή η Heap_1 είναι ντετερμινιστική.

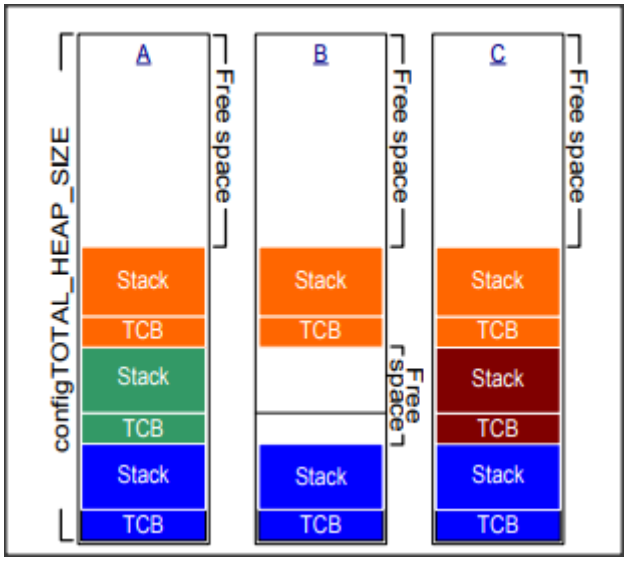
Κατά το compilation δεσμεύεται ένας πίνακας C με μέγεθος "**configTOTAL_HEAP_SIZE**" που ορίζεται στο αρχείο *FreeRTOSConfig.h* ο οποίος αποτελεί τον σωρό του FreeRTOS. Ο πίνακας αυτός υποδιαιρείται σε μικρότερα block τα οποία εκχωρούνται με διαδοχικές κλήσεις στην pvPortMalloc().Αρχικά η εφαρμογή θα φαίνεται να καταναλώνει μεγάλη ποσότητα RAM, λόγω του πίνακα σωρού, αν και ακόμα δεν έχει εκχωρηθεί κάποιο block μνήμης. Το σχήμα 2.4.2 δείχνει στιγμιότυπα του σωρού κατά την διαδοχική δημιουργία task.



Σχήμα 2.4.2 :Ο άδειος σωρός,ύπαρξη 1 task και 3 task[6].

Heap_2
Εδώ υλοποιείται και η συνάρτηση vPortFree() και επομένως είναι κατάλληλη για εφαρμογές που δημιουργούν και διαγράφουν αντικείμενα πυρήνα δυναμικά κατά την διάρκεια ζωής τους. Όπως για παράδειγμα στην περίπτωση που δημιουργούμε και διαγράφουμε task δυναμικά κατά την διάρκεια εκτέλεσης της εφαρμογής. Η Heap_2 υλοποιεί αλγόριθμο best-fit έτσι ώστε όταν ζητείται ένα block μνήμης να παραχωρηθεί το μικρότερο δυνατό ελεύθερο block σωρού. Και πάλι ο πίνακας σωρού δεσμεύεται στατικά με το μέγεθος που ορίζεται στο

FreeRTOSConfig.h, επομένως αρχικά η εφαρμογή φαίνεται να καταναλώνει πολλή RAM. Η Heap_2 δεν υλοποιεί συνένωση(coalescing) γειτονικών ελεύθερων block συνεπώς είναι επιρρεπής στον κατακερματισμό. Βέβαια αν τα block που ζητούνται και διαγράφονται είναι του ίδιου μεγέθους, όπως στην περίπτωση της δημιουργίας και διαγραφής task, δεν υπάρχει τέτοιος κίνδυνος. Γενικά στην θέση της Heap_2 προτείνεται η χρήση της Heap_4 η οποία προσφέρει βελτιωμένη λειτουργικότητα, συμπεριλαμβανομένου και του coalescing. Τέλος η Heap_2 δεν είναι ντετερμινιστική αλλά είναι ταχύτερη από τις περισσότερες υλοποιήσεις των C standard malloc() και free(). Στο σχήμα 2.4.3 βλέπουμε στιγμιότυπα του σωρού πριν και μετά την διαγραφή ενός task και δημιουργία ενός νέου.



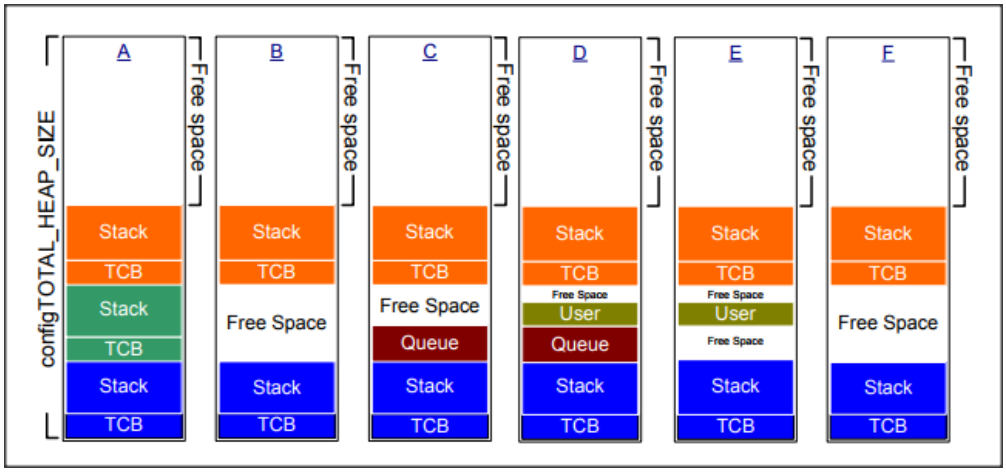
Σχήμα 2.4.3 :Σωρός με 3 task,διαγραφή ενός και δημιουργία νέου[6].

Heap_3

Εδώ οι pvPortMalloc() και vPortFree() χρησιμοποιούν τις standard malloc() και free() κι επομένως το μέγεθος σωρού καθορίζεται από το linker script και όχι από το *FreeRTOSConfig.h*. Επιπλέον με την απενεργοποίηση του scheduler πριν την κλήση των malloc() και free() στην Heap_3 εξασφαλίζεται ότι δεν θα συμβεί κάποιο context switch πρωτού ολοκληρωθεί η εκτέλεση τους, γεγονός που θα μπορούσε να οδηγήσει τις δομές διαχείρισης μνήμης σε ασυνέπεια.

Heap_4

Όπως στην Heap_1 και Heap_2 η μνήμη παραχωρείται από ένα στατικά εκχωρημένο πίνακα σωρού με μέγεθος "configTOTAL_HEAP_SIZE". Η Heap_4 υλοποιεί την vPortFree() και για την pvPortMalloc() υλοποιεί αλγόριθμο first-fit, δηλαδή θα παραχωρηθεί το πρώτο ελεύθερο block σωρού το οποίο είναι αρκετά μεγάλο. Επίσης έχουμε και coalescing γειτονικών ελεύθερων block σε ένα ενιαίο μεγάλο block. Επομένως η Heap_4 είναι κατάλληλη για εφαρμογές που ζητούν και διαγράφουν block μνήμης διαφορετικού μεγέθους αφού αντιμετωπίζεται ο κίνδυνος κατακερματισμού. Μια επιπλέον δυνατότητα που δίνεται είναι η δυνατότητα ορισμού της διεύθυνσης του πίνακα σωρού ώστε να μπορεί να τοποθετηθεί,για παράδειγμα, σε μια γρηγορότερη εσωτερική μνήμη του συστήματος αντί σε μια πιο αργή εξωτερική. Τέλος,όμοια με την Heap_2, η Heap_4 δεν είναι ντετερμινιστική αλλά γρηγορότερη από τις περισσότερες υλοποιήσεις malloc() και free(). Το σχήμα 2.4.4 αποτελεί ένα πλήρες παράδειγμα δυναμικής διαχείρισης μνήμης.



Σχήμα 2.4.4 :Σωρός με 3 task,διαγραφή ενός,παραχώρηση μνήμης για ένα Queue,παραχώρηση μνήμης από αίτημα χρήστη,ελευθέρωση του Queue block,ελευθέρωση του block χρήστη coalescing με το πρώην Queue block σε ένα ενιαίο ελεύθερο block[6].

Heap_5

Η Heap_5 είναι όμοια από άποψη λειτουργικότητας με την Heap_4 ωστόσο εδώ δίνεται η δυνατότητα παραχώρησης μνήμης από μη συνεχόμενα τμήματα διευθύνσεων. Στα προηγούμενα ο σωρός ήταν ένας πίνακας C δηλαδή ένας συνεχόμενος χώρος διευθύνσεων ο οποίος με διαδοχικές κλήσεις χωρίζονταν σε δεσμευμένα block. Εδώ ωστόσο υπάρχει η δυνατότητα με την κλήση της API συνάρτησης vPortDefineHeapRegions() να ορίσουμε διακριτές μη συνεχόμενες περιοχές μνήμης από τις οποίες θα γίνεται η δυναμική παραχώρηση μνήμης. Η συνάρτηση vPortDefineHeapRegions() αναμένει σαν όρισμα έναν δείκτη σε πίνακα των C δομών που φαίνονται στο σχήμα 2.4.5.

```
typedef struct HeapRegion
{
    /* The start address of a block of memory that will be part of the heap.*/
    uint8_t *pucStartAddress;

    /* The size of the block of memory in bytes. */
    size_t xSizeInBytes;

} HeapRegion_t;
```

Σχήμα 2.4.5 :Η δομή HeapRegion[6].

Προφανώς μια τέτοια δυνατότητα δίνει σχεδιαστική ευελιξία σε συστήματα με κάποια ανομοιομορφία στο χάρτη μνήμης και θα πρέπει να προτιμάται μόνο όταν υπάρχει ανάγκη για κάτι τέτοιο.

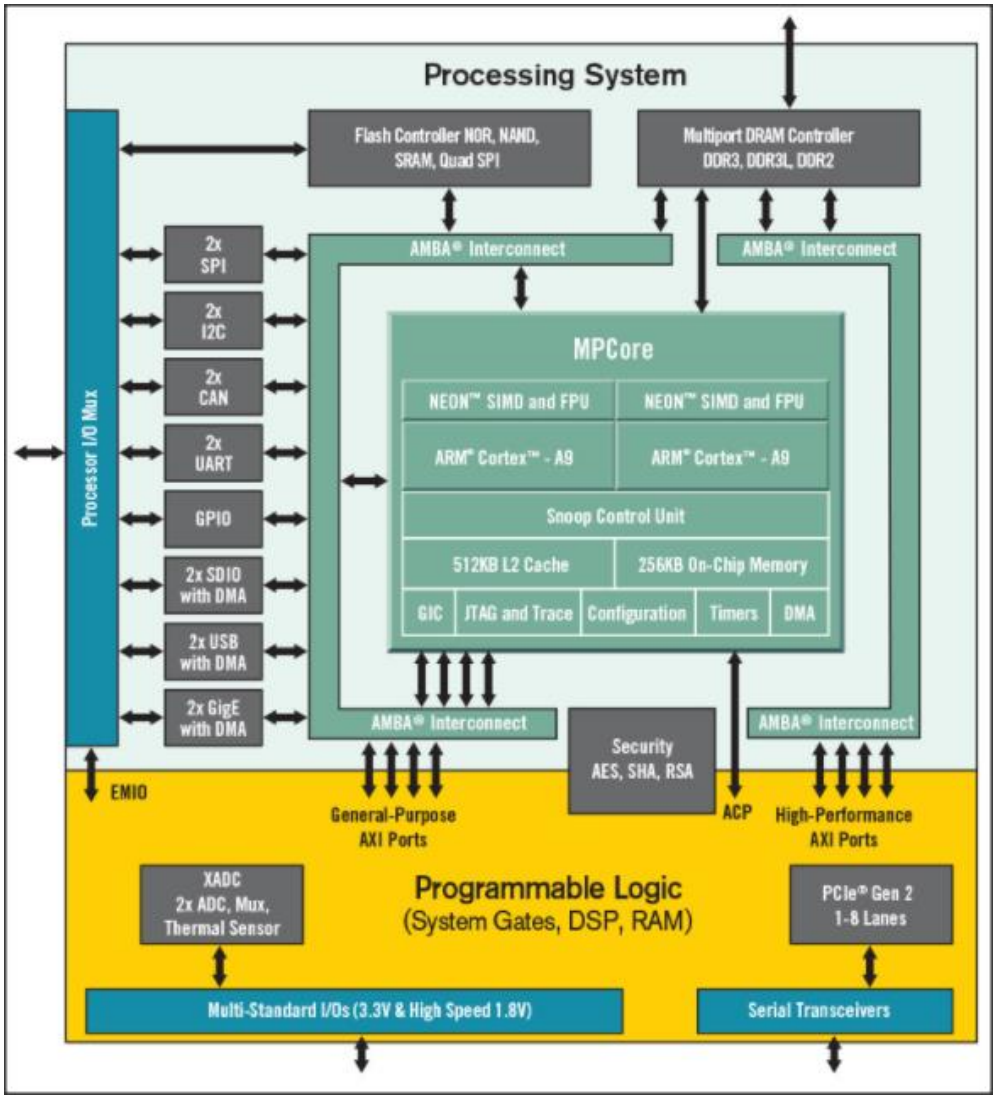
Συμπερασματικά,βλέπουμε πως δίνονται αρκετές επιλογές για την δυναμική διαχείριση μνήμης με βασικότερες την απλή και φθηνή ,από άποψη μεγέθους και πολυπλοκότητας, Heap_1 και την πλούσια, από άποψη λειτουργιών,Heap_4. Σε εφαρμογές, αυστηρά πραγματικού χρόνου (hard-real time) ή εφαρμογές με αυστηρά κριτήρια ασφαλείας δεν επιτρέπεται η δυναμική παραχωρήση μνήμης λόγω της αβεβαιότητας που εισάγεται με τον μη-ντετερμινισμό, τον κατακερματισμό της μνήμης κι αποτυχημένες παραχωρήσεις μνήμης. Για ένα τέτοιο σύστημα λοιπόν,μετά από καθορισμό των δομών και αντικειμένων πυρήνα που απαιτούνται (πόσα task για παράδειγμα), προχωρούμε σε δημιουργία όλων των αντικειμένων πριν την εκκίνηση του scheduler κι έτσι η Heap_1 είναι η καταλληλότερη επιλογή. Αν, από την άλλη, το υπό σχεδιασμό σύστημα δεν έχει τόσο αυστηρές χρονικές προδιαγραφές και προτιμάται η ευελιξία της παραχώρησης και ελευθέρωσης μνήμης σε οποιοδήποτε σημείο η Heap_4 θα ήταν μια καλή επιλογή. Τέλος, πέρα από τον πυρήνα του FreeRTOS οι pvPortMalloc() και vPortFree() μπορούν να κληθούν κι από τον κώδικα χρήστη στη θέση των malloc() και free() αφού έχουν όμοια prototypes.

3 *Η αναπτυξιακή πλακέτα Digilent Zybo*

Η αναπτυξιακή πλακέτα Zybo της Digilent αποτελεί ένα πολύ ισχυρό, από άποψη δυνατοτήτων, περιβάλλον ανάπτυξης όπως θα γίνει εμφανές από την παρουσίαση των χαρακτηριστικών του. Ταυτοχρόνα αποτελεί και μία από τις πιο οικονομικές λύσεις ανάμεσα στα αναπτυξιακά του Xilinx Zynq-7000 SoC το οποίο είναι ένα αρκετά δημοφιλές SoC, χάρη στην ενσωμάτωση πάνω στο ίδιο chip δύο ισχυρών πυρήνων ARM και FPGA. Συνεπώς η χρήση του FreeRTOS στο Zybo αποτελεί μια δόκιμη επιλογή καθώς ο συνδυασμός των δυνατοτήτων του hardware σε συνδυασμό με το FreeRTOS software μπορεί να αποτελέσει την βάση δημιουργίας πολύ πλούσιων εφαρμογών στο μέλλον κι συνεπώς η παρούσα εργασία να αποτελέσει την βάση σημαντικών μελλοντικών επεκτάσεων.

3.1 Τεχνικά χαρακτηριστικά του Zybo

*Σχήμα 3.1.1 :*Το Xilinx Zynq-700 All Programmable SoC το οποίο περιέχεται στο Zybo.



Σχήμα 3.1.2 : Xilinx Zynq 7000 All Programmable SoC Processing System[7].

Στο Zynq 7000 η Xilinx ενσωμάτωσε 2 cores ARM Cortex-A9 σαν PS (Processing System) καθώς και FPGA 28nm (Field Programmable Gate Array) PL (Programmable Logic ή Fabric).Επιπλέον ενσωματώνεται ένα πλήθος περιφερειακών συσκευών καθώς και δίαυλοι υψηλού εύρους ζώνης. Συγκεκριμένα:

- **Processing System (PS)**

  - **2 cores ARM Cortex A9**
    - 2.5DMIPS/MHZ/core,1GHz max Cpu Frequency, Coherent
    - Multiprossecor Support

  - **Caches**
    - 32 KB Level 1 4-way-set-associative instruction and data(per CPU)
    - 512 KB Level 2 8-way-set-associative(shared)

  - **Memory**
    - On-chip ROM,On-chip 256KB RAM
    - DDR3,DDR3L,DDR2,LPDDR2 external memory support
    - 2x QUAD-SPI,NAND,NOR external Flash memory support

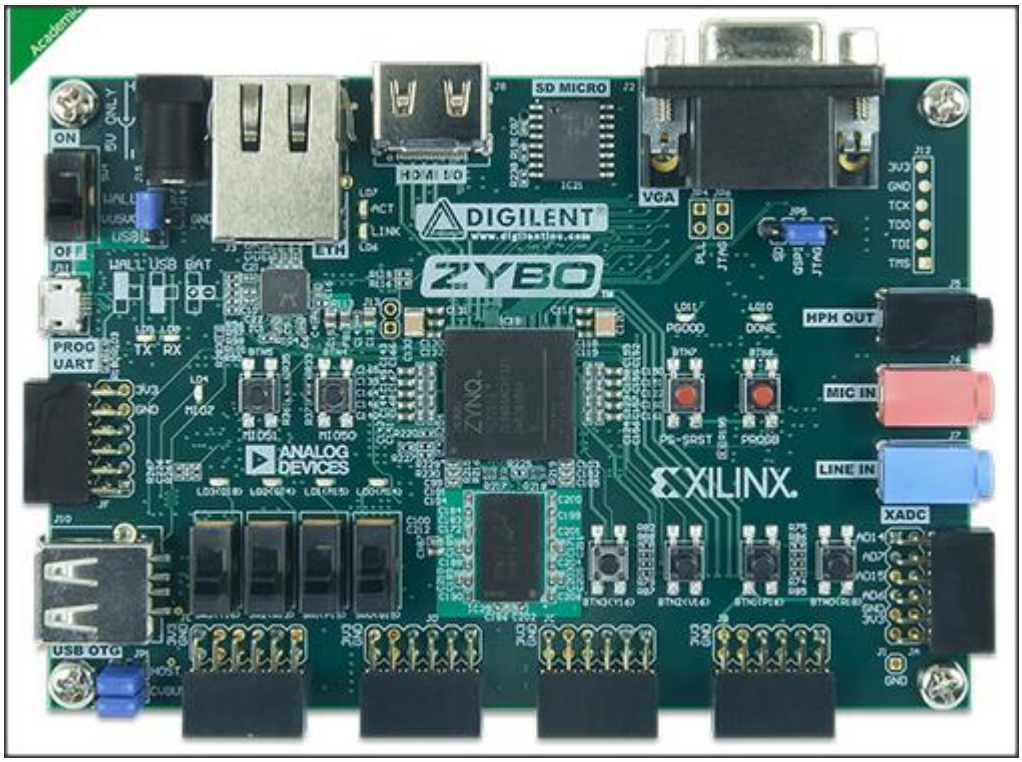
- **Peripherals**
  - 2x UART,2x CAN 2.0B,2x I2C,2x SPI,4x 32b GPIO,2x USB 2.0,2x Gigabit Ethernet,2x SD/SDIO

- **Security**
  - RSA Authentication,AES και SHA 256-bit Decryption and
  - Authentication for secure boot

- **PS to PL Interface Ports**
  - 2x AXI 32b Master 2x AXI 32-bit Slave
  - 4x AXI 64-bit/32-bit Memory
  - AXI 64-bit ACP
  - 16 interrupts

❖ **Programmable Logic (PL)**

- Artix 7 FPGA
  - 28,000 logic cells
  - 240 KB Block RAM
  - 80 DSP slices
- On-chip dual channel,12-bit,1MSPS analog-to-digital converter
- (XADC)



Σχήμα 3.1.3 :Zybo Zynq 7000 Trainer Board[8].

Χαρακτηριστικά του Zybo

- **Programming**
  - On board JTAG and UART to USB converter

- **I/O**
  - 6 PMOD ports
  - Audio codec with headphone out, microphone,and line in jacks
  - Gigabit Ethernet
  - USB 2.0
  - HDMI port INPUT/OUTPUT
  - 16-bit VGA

- **USER INTERFACE**
  - 4x Switches
  - 6x Buttons
  - 5x LEDs

- **STORAGE**
  - SD Card slot

3.2 Εγκατάσταση του FreeRTOS στο Zybo

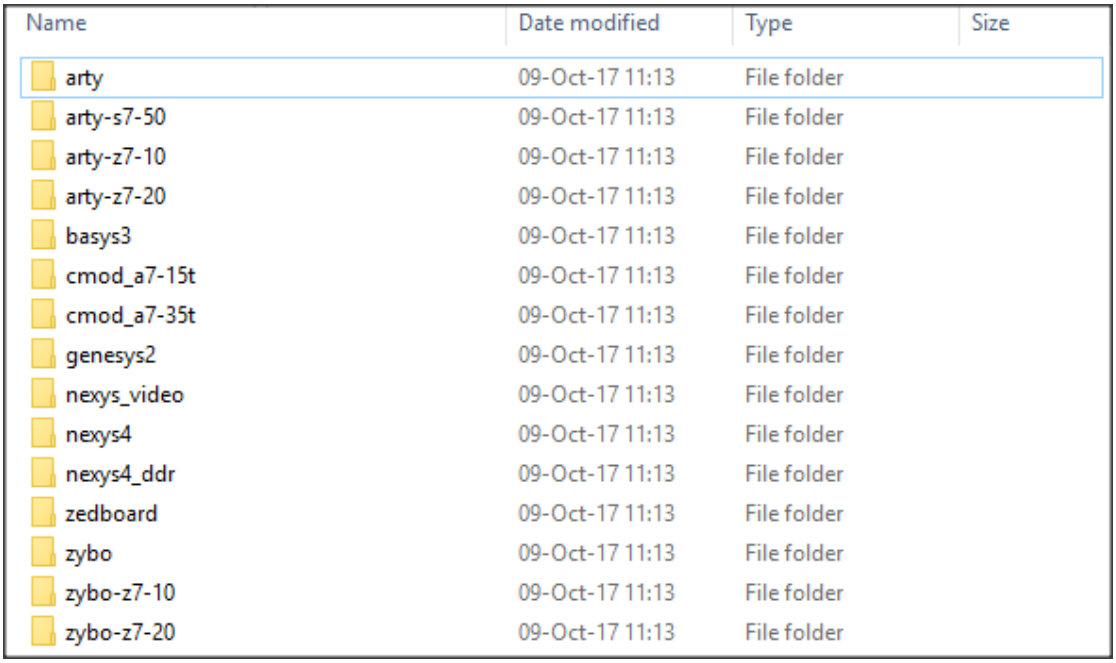
Αν κοιτάξουμε στο φάκελο Demo του FreeRTOS θα δούμε τον υποφάκελο Cortex_A9_Zynq_ZC702.

Σχήμα 3.2.1 :Φάκελος με το Port του FreeRTOS για το Xilinx Zynq 7000.

Πρόκειται για ένα έτοιμο project για το Xilinx SDK αλλά για την αναπτυξιακή πλακέτα ZC702 που κατασκευάζει η Xilinx κι όχι για το Zybo. Ο προγραμματισμός του Zybo απαιτεί και προγραμματισμό του FPGA ώστε να "συνδεθούν" τα Led και οι διακόπτες της αναπτυξιακής πλακέτας με τον ARM επεξεργαστή κι έτσι να υπάρχει έλεγχος μέσω του κώδικα.

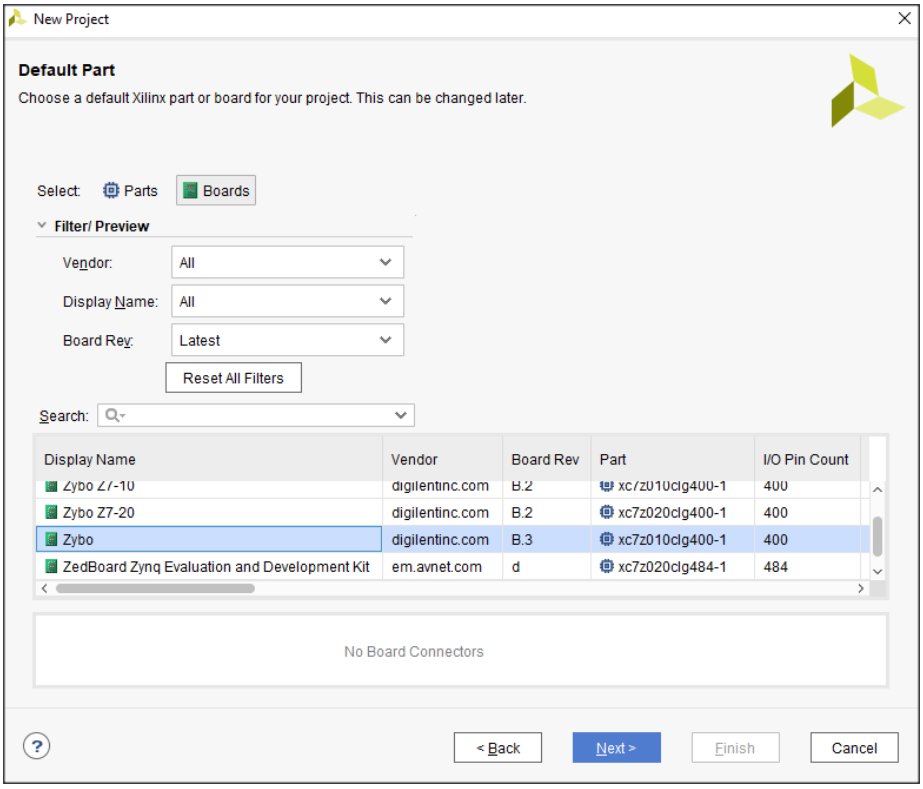
<u>Δημιουργία αρχείου Hardware Design και bitstream στο Vivado 2017.3 [9]</u>

1. Η Digilent παρέχει ένα φάκελο αρχείων ,για κάθε αναπτυξιακή της πλακέτα, με σκοπό να ενσωματωθεί στο Vivado κι έτσι να διευκολυνθεί ο σχεδιαστής. Συγκεκριμένα αυτά τα XML αρχεία περιγράφουν τις διεπαφές εισόδου εξόδου GPIO (για παράδειγμα διακόπτες, LED, USB-UART, DDR Memory,Ethernet κλπ.) στην πλακέτα κι έτσι διευκολύνεται η αρχική παραμετροποίηση ως επίσης και η χρήση έτοιμων GPIO IP block στο block design tool του Vivado. Αντιγράφουμε λοιπόν τα αρχεία αυτά στο φάκελο board_files της εγκατάστασης του Vivado.

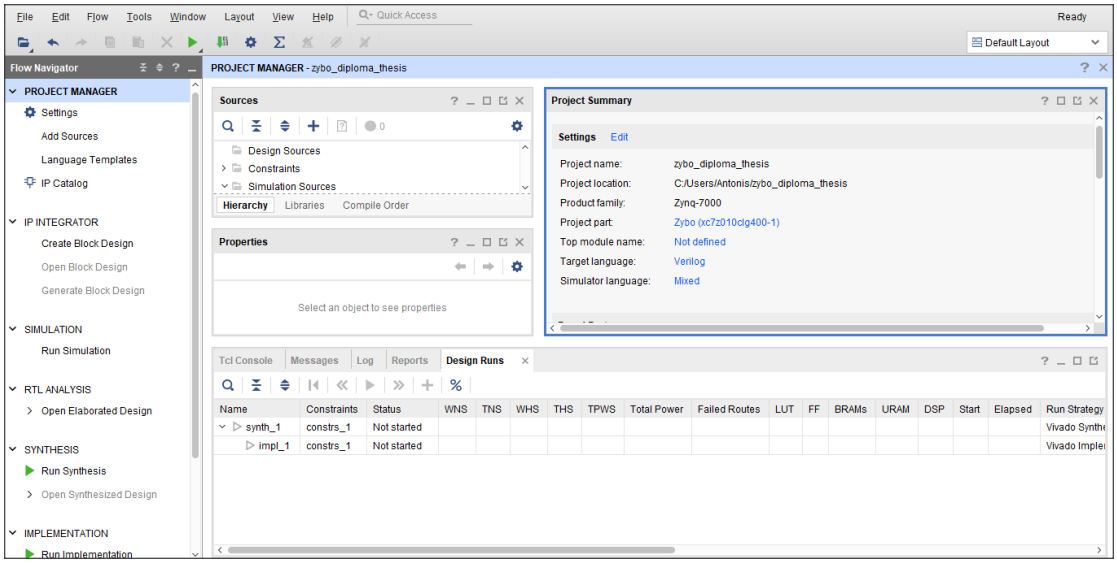


Σχήμα 3.2.2 :Οι φάκελοι με τα αρχεία κάθε αναπτυξιακής πλακέτας της Digilent.

2. Έπειτα δημιουργούμε ένα νέο RTL Project στο Vivado κι επιλέγουμε το Zybo στη καρτέλα Default Part/Boards

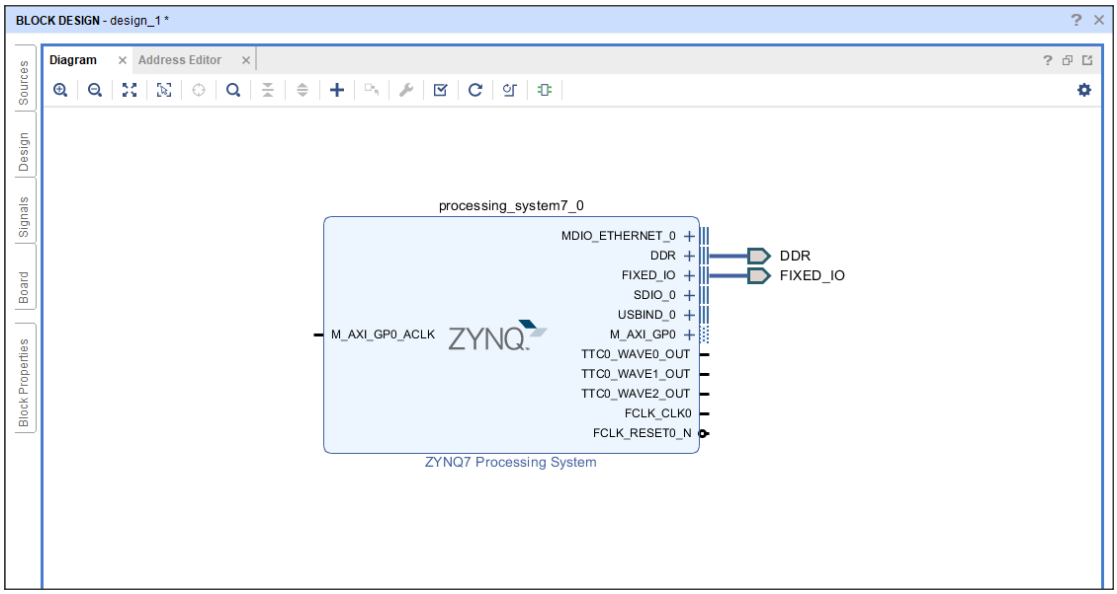


Σχήμα 3.2.3 :Η καρτέλα επιλογής πλακέτας.



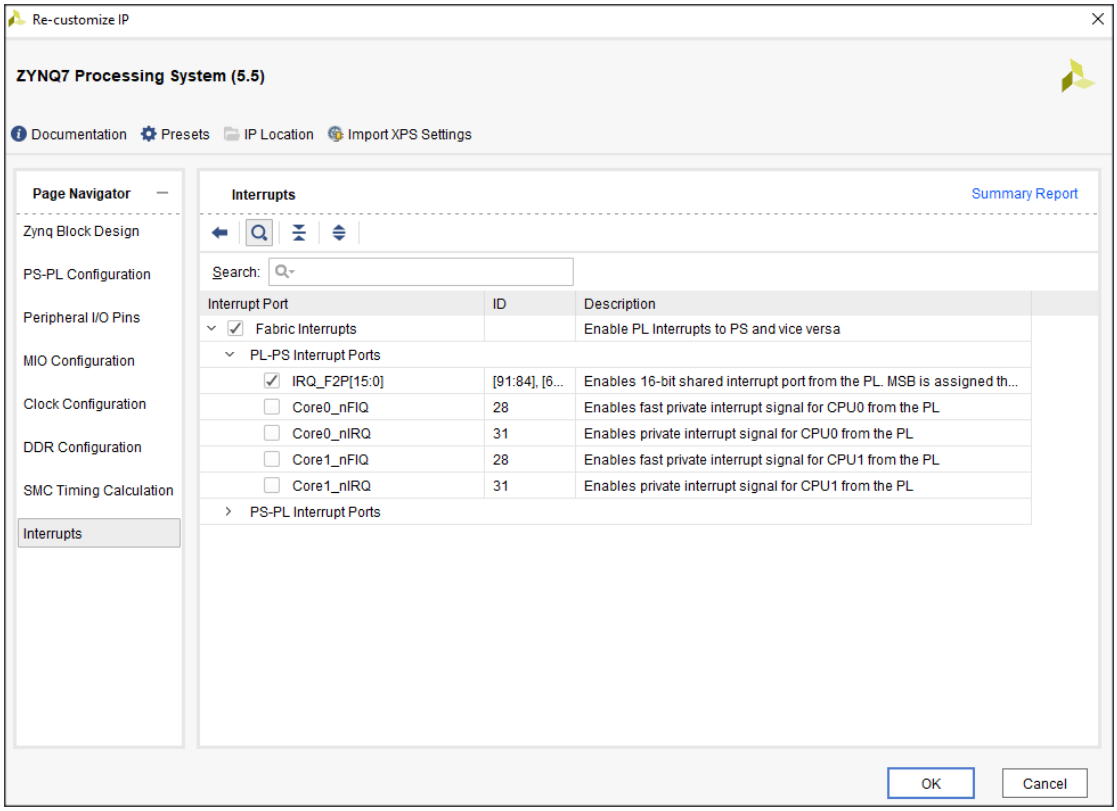
Σχήμα 3.2.4 :Το ανοικτό Project στο Vivado.

3. Επιλέγουμε *Create Block Design,Add IP...ZYNQ7 Processing System* και *Run Block Automation*



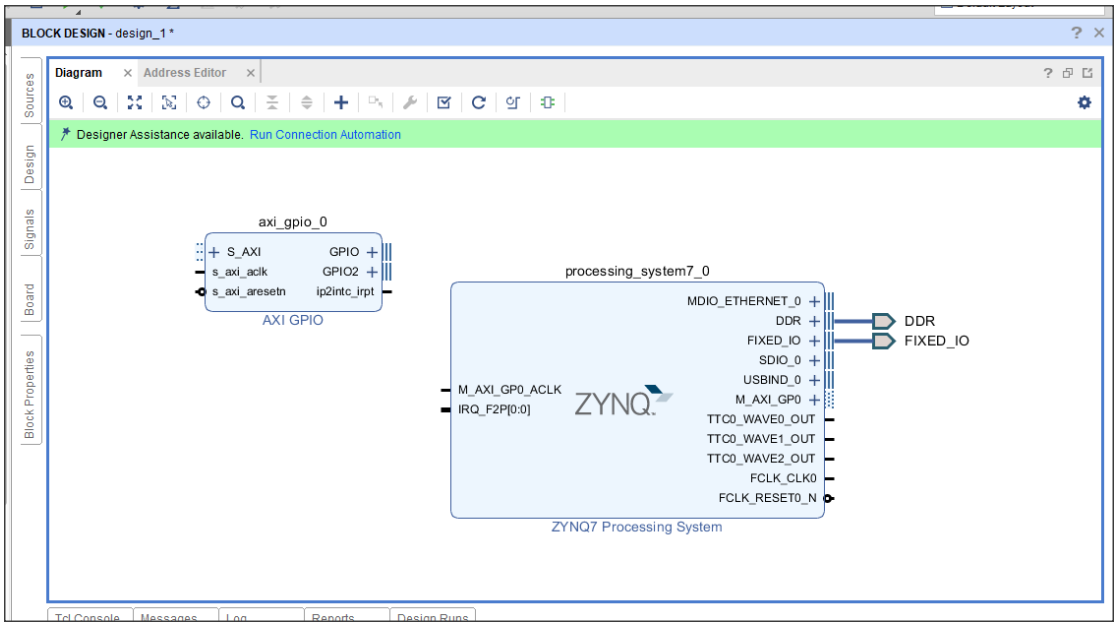
Σχήμα 3.2.5 :Το ZYNQ7 IP block.

4. Στην συνέχεια επιλέγουμε το ZYNQ block κι ενεγοποιούμε τα interrupts:



Σχήμα 3.2.6 :Ενεργοποίηση των PL interrupt.

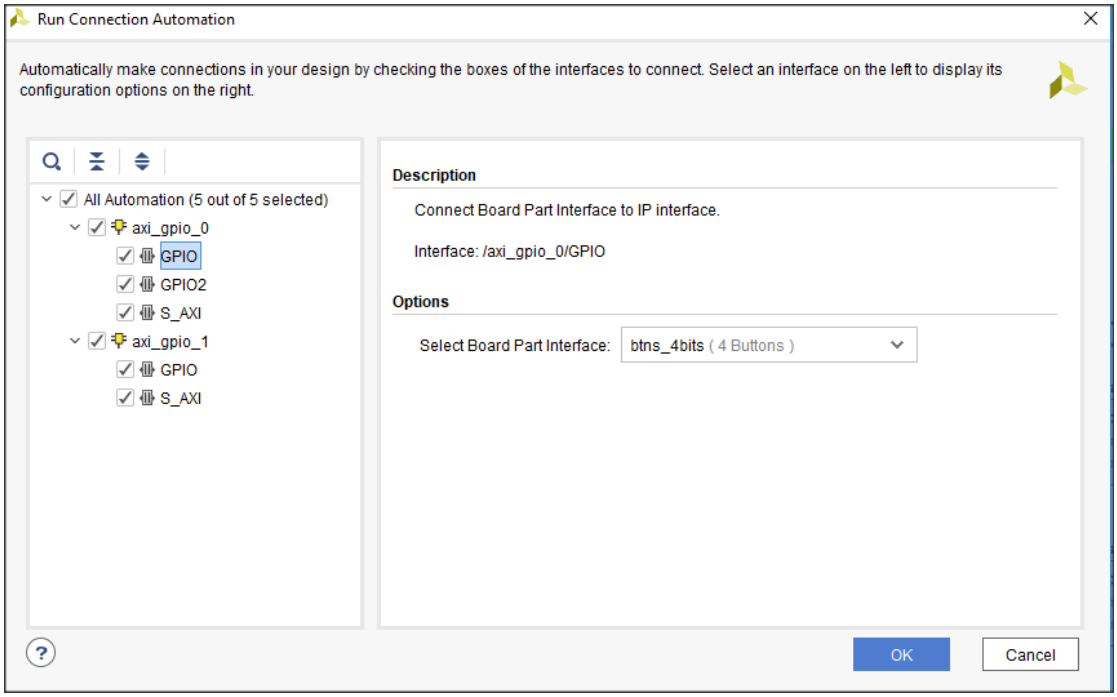
5. Προσθέτουμε AXI GPIO IP block και επιλέγουμε *Enable Dual Channel* καθώς *Enable Interrupt:*



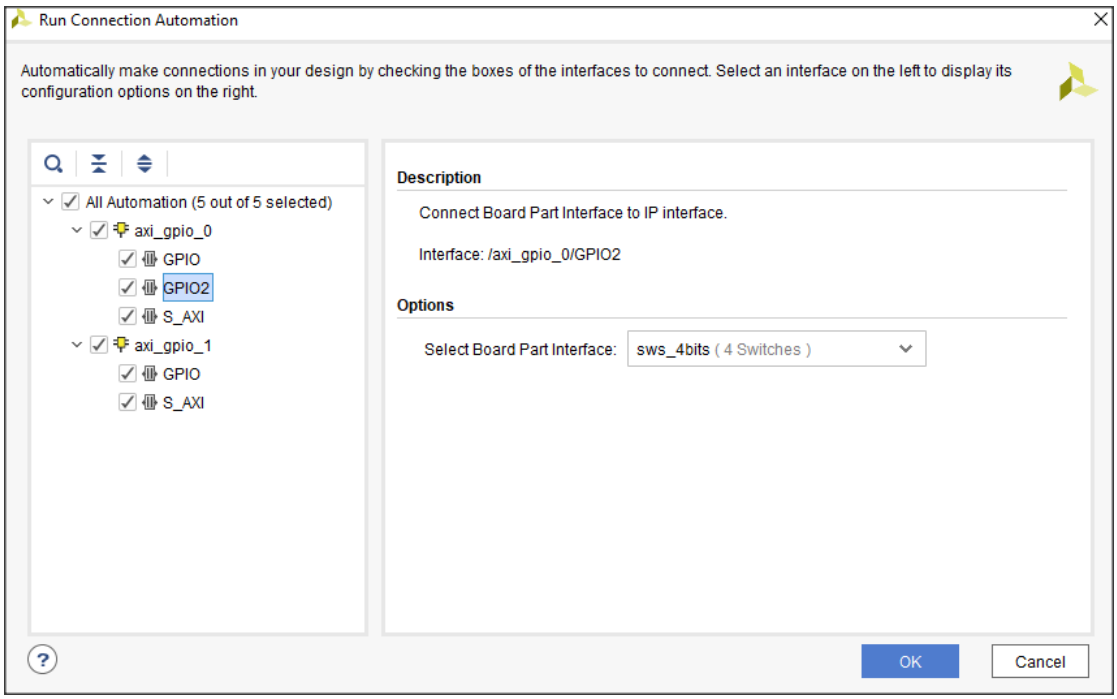
Σχήμα 3.2.7 :Τα AXI GPIO block που προσθέσαμε.

Στην συνέχεια προσθέτουμε ένα ακόμα AXIP GPIO block χωρίς όμως να κάνουμε τις επιλογές *Enable Dual Channel* και *Enable Interrupt.*

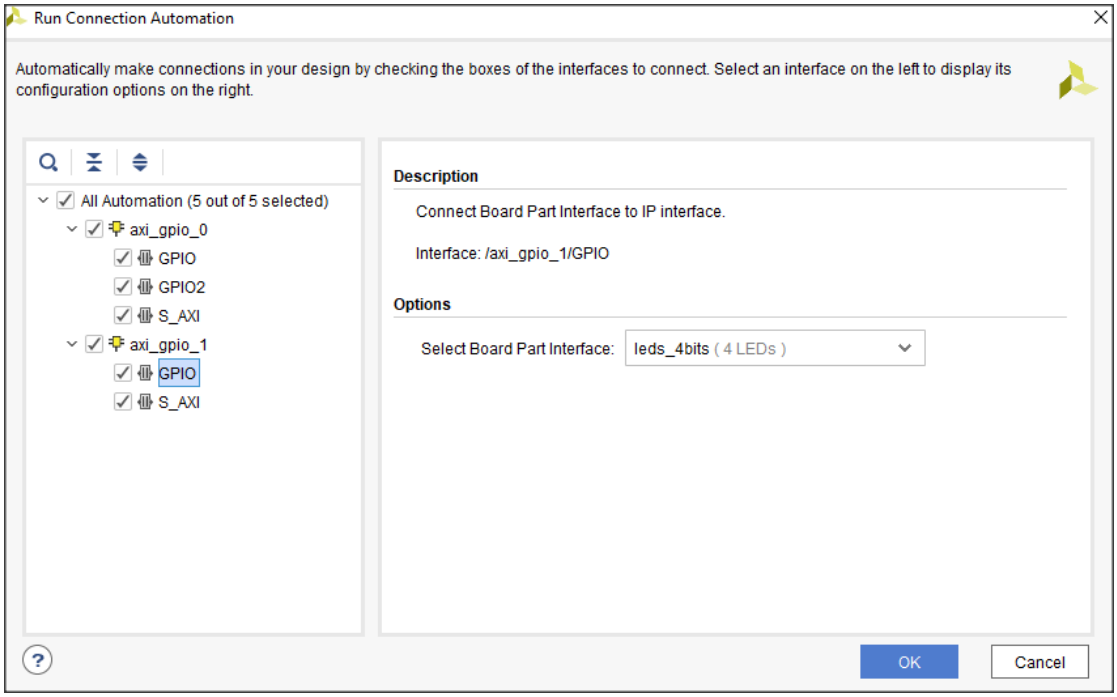
6. Πατάμε *Run Connection Automation* και κάνουμε τις ακόλουθες επιλογές:



Σχήμα 3.2.8 : Σύνδεση με τα 4 push buttons.

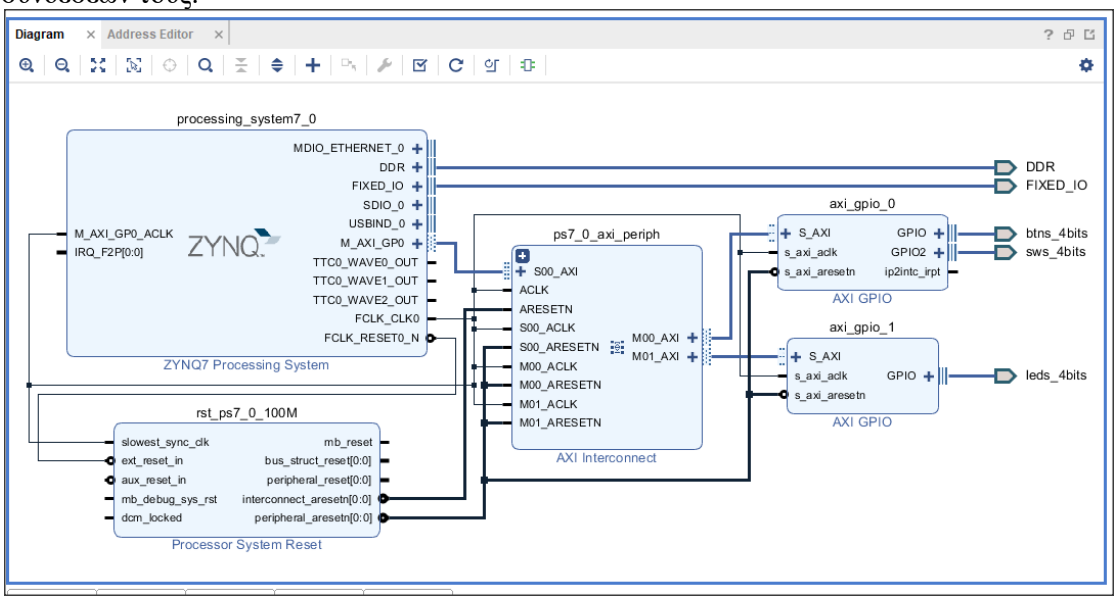


Σχήμα 3.2.9 :Σύνδεση με τα 4 slide switches.



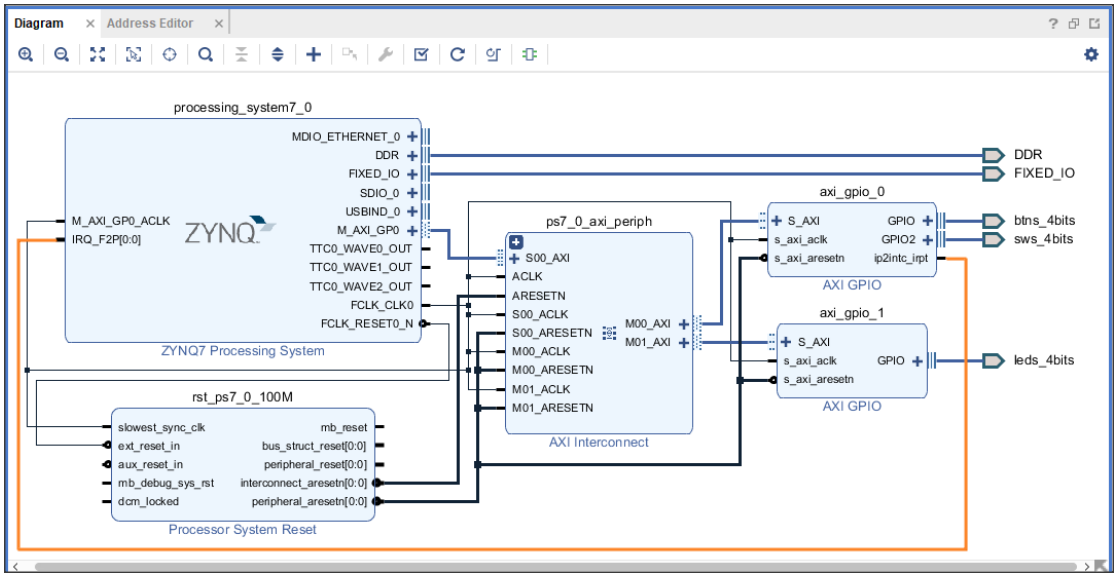
Σχήμα 3.2.10 :Σύνδεση με τα 4 Leds.

Στην συνέχεια και μετά από Regenerate Layout έχουμε το παρακάτω σχήμα block και συνδέσεων τους:



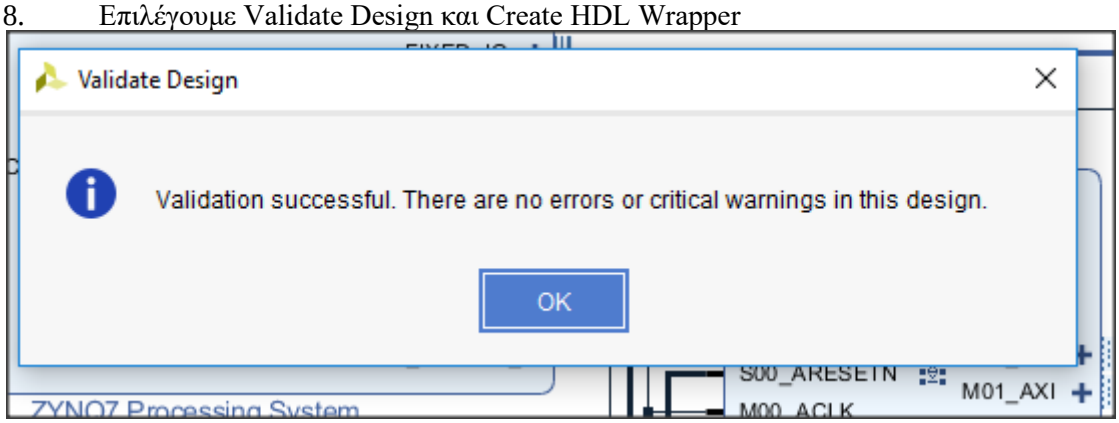
Σχήμα 3.2.11 :Το διάγραμμα λογικών συνδέσεων.

7. Έπειτα χειροκίνητα συνδέουμε και την γραμμή των interrupt:

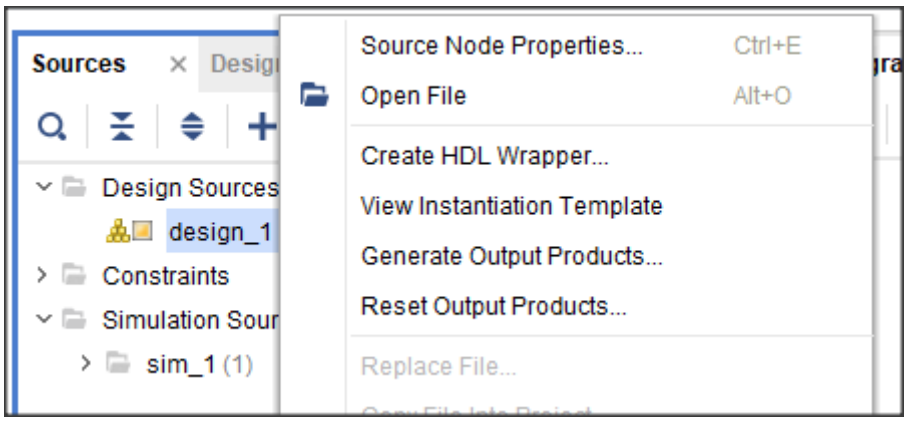


Σχήμα 3.2.12 :Το πλήρες λογικό διάγραμμα συνδέσεων με την γραμμή interrupt.

8. Επιλέγουμε Validate Design και Create HDL Wrapper

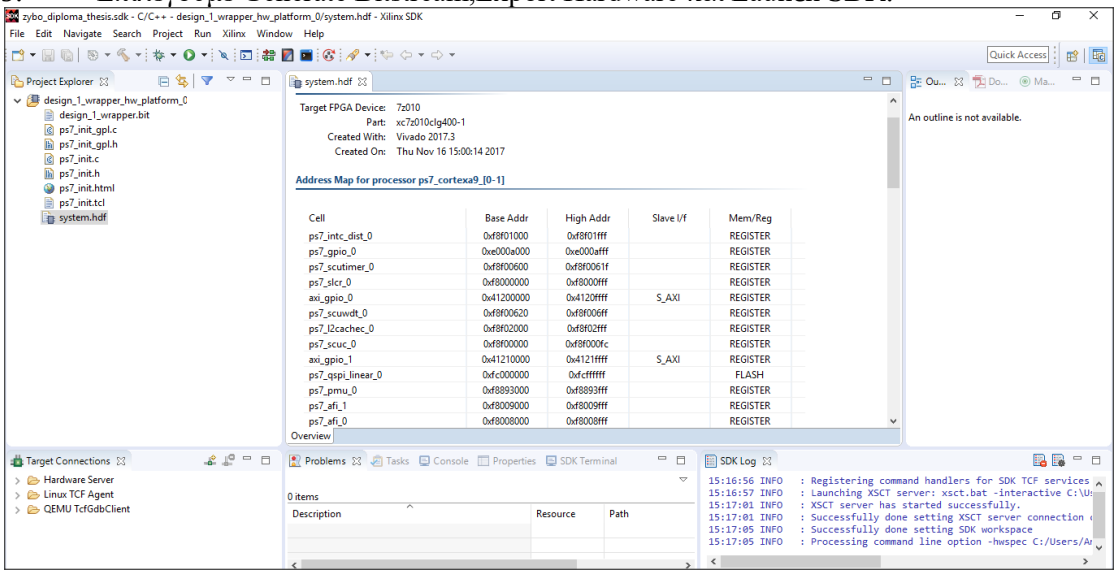


Σχήμα 3.2.13 :Validate Design.



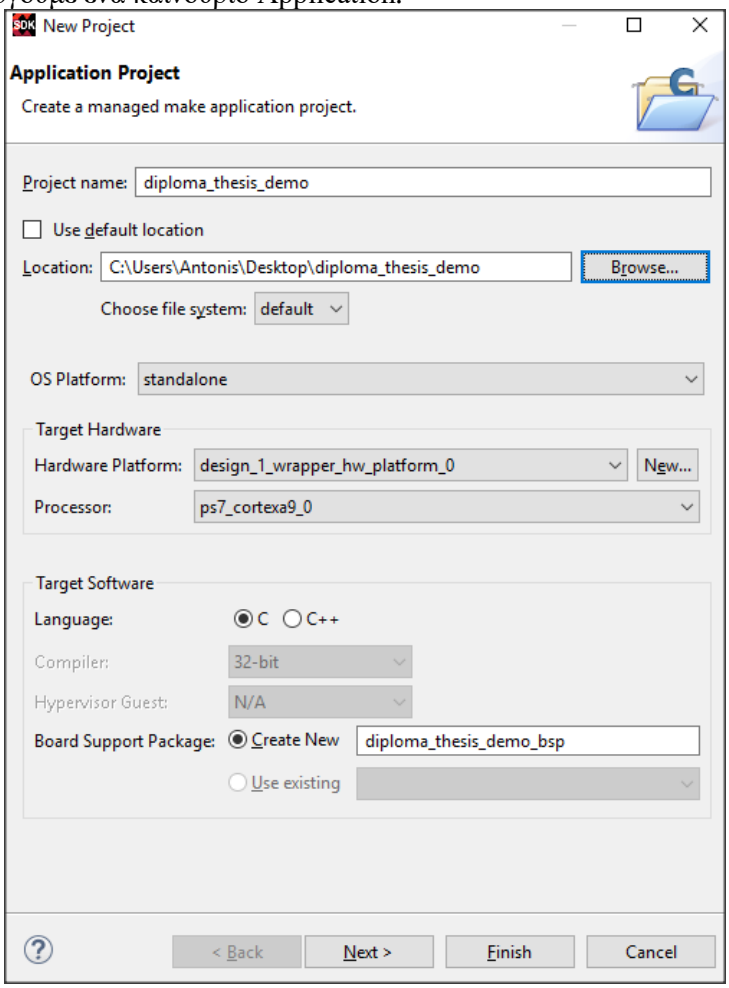
Σχήμα 3.2.14 :Create HDL Wrapper.

9. Επιλέγουμε Generate Bitstream,Export Hardware και Launch SDK.

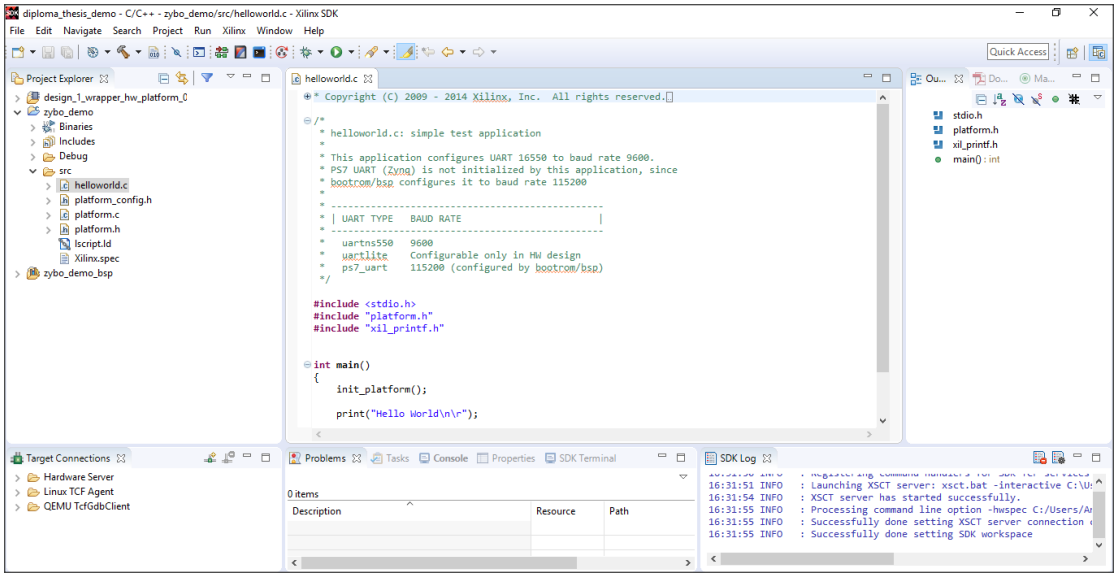


Σχήμα 3.2.15 :Το Xilinx SDK με το Hardware platform που δημιουργήθηκε στο Vivado.

10. Δημιουργούμε ένα καινούριο Application.

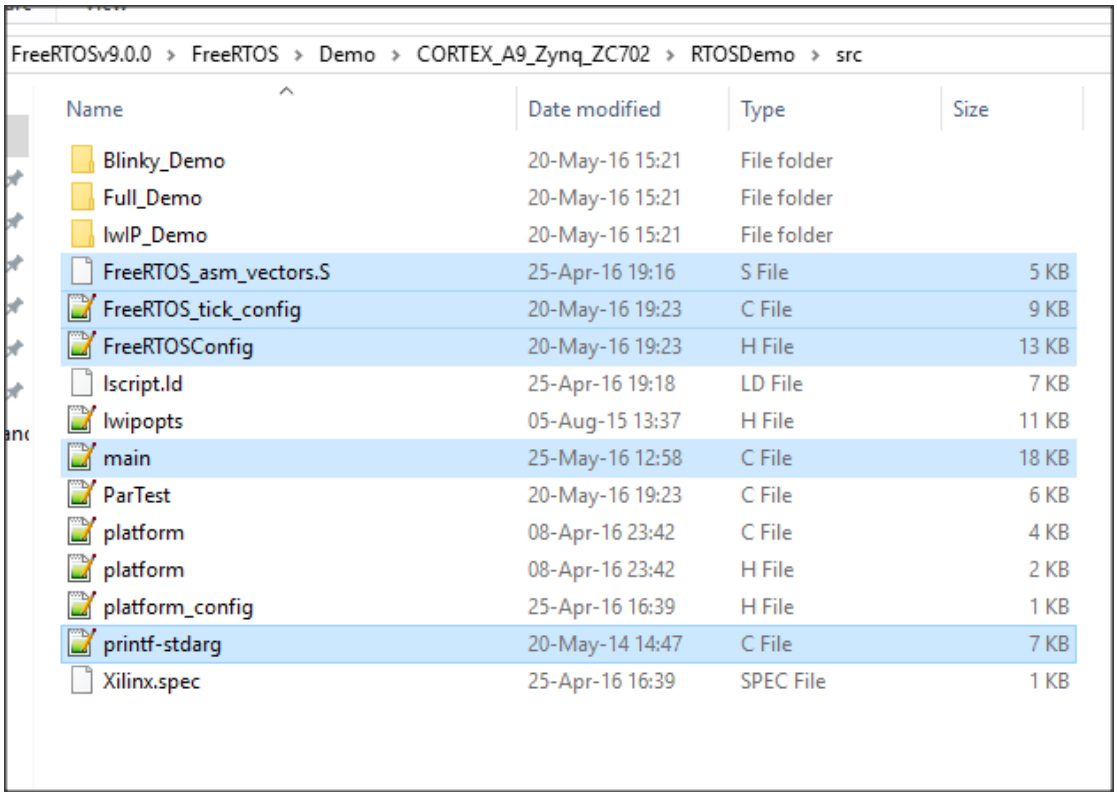


Σχήμα 3.2.16 :Στην επιλογή OS Platform επιλέγουμε standalone καθώς τα απαραίτητα αρχεία για το FreeRTOS θα δοθούν ως κώδικας του project, στο target hardware έχουμε το Hardware Platform που φτιάχτηκε στο Vivado

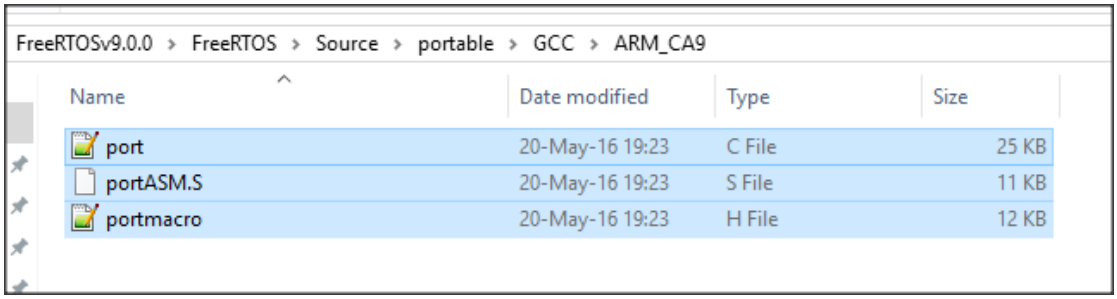


Σχήμα 3.2.17 :To default project Helloworld.

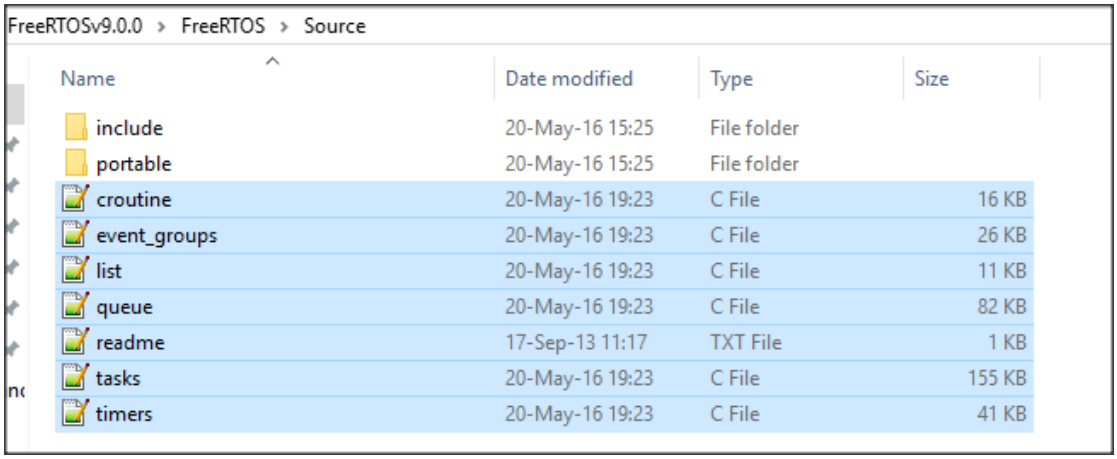
11. Βρίσκουμε τα απαραίτητα αρχεία για την χρήση του FreeRTOS και τα εισάγουμε στο φάκελο Project src.



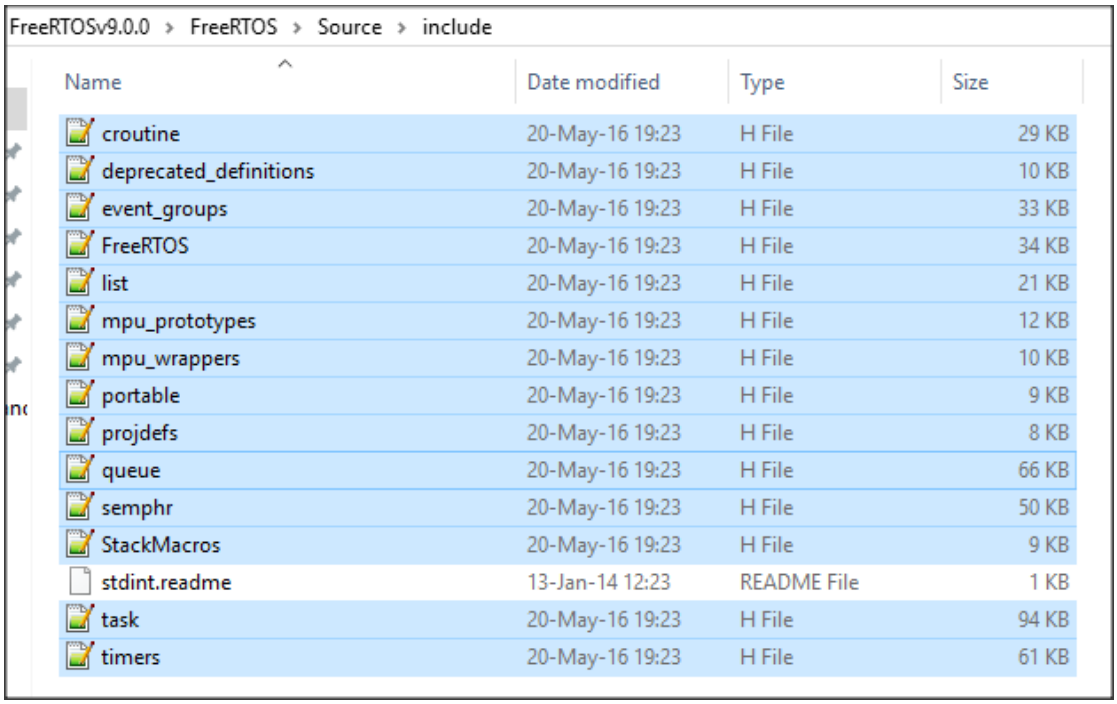
Σχήμα 3.2.18 : Τα αρχεία από το demo.



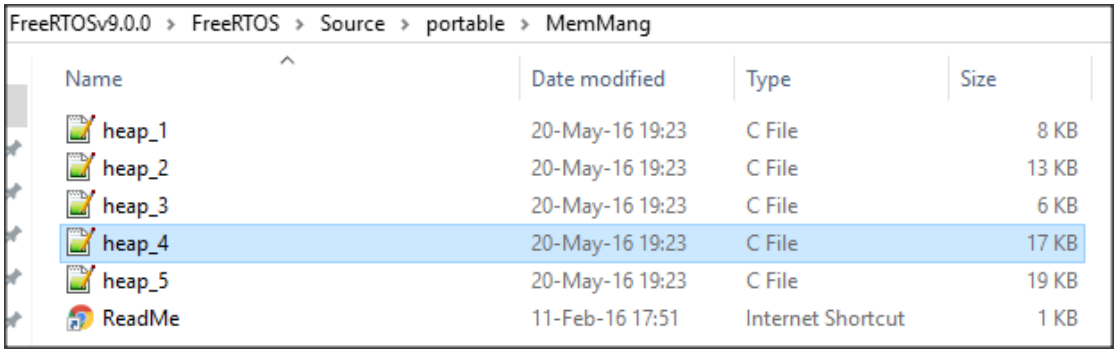
Σχήμα 3.2.19 :Port architecture specific αρχεία.



Σχήμα 3.2.20 :Τα βασικά source αρχεία του FreeRTOS που είναι κοινά για όλα τα port.



Σχήμα 3.2.21 :Τα header αρχεία του FreeRTOS.



Σχήμα 3.2.22 : Επιλέγουμε το heap_4 για την διαχείριση μνήμης.

12. Προσθέτουμε στο linker script την ακόλουθη γραμμή

lscript.ld

```
{
   ps7_ddr_0 : ORIGIN = 0x100000, LENGTH = 0x1FF00000
   ps7_qspi_linear_0 : ORIGIN = 0xFC000000, LENGTH = 0x1000000
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
}

/* Specify the default entry point to the program */

ENTRY(_vector_table)

/* Define the sections, and where they are mapped in memory */

SECTIONS
{
.text : {
   *(.freertos_vectors)
   KEEP (*(.vectors))
   *(.boot)
   *(.text)
   *(.text.*)
   *(.gnu.linkonce.t.*)
   *(.plt)
   *(.gnu_warning)
   *(.gcc_execpt_table)
   *(.glue_7)
   *(.glue_7t)
   *(.vfp11_veneer)
   *(.ARM.extab)
   *(.gnu.linkonce.armextab.*)
} > ps7_ddr_0
```

Σχήμα 3.2.23 :Linker script.

Έτσι ώστε να τοποθετηθεί στην αρχή ο interrupt vector table με τους handlers του FreeRTOS που ορίζεται στο αρχείο FreeRTOS_asm_vectors.S

13. Τέλος κάνουμε comment την παρακάτω γραμμή εφόσον δεν επιθυμούμε ο κώδικας του task να τρέξει σε ARM THUMB mode.

port.c

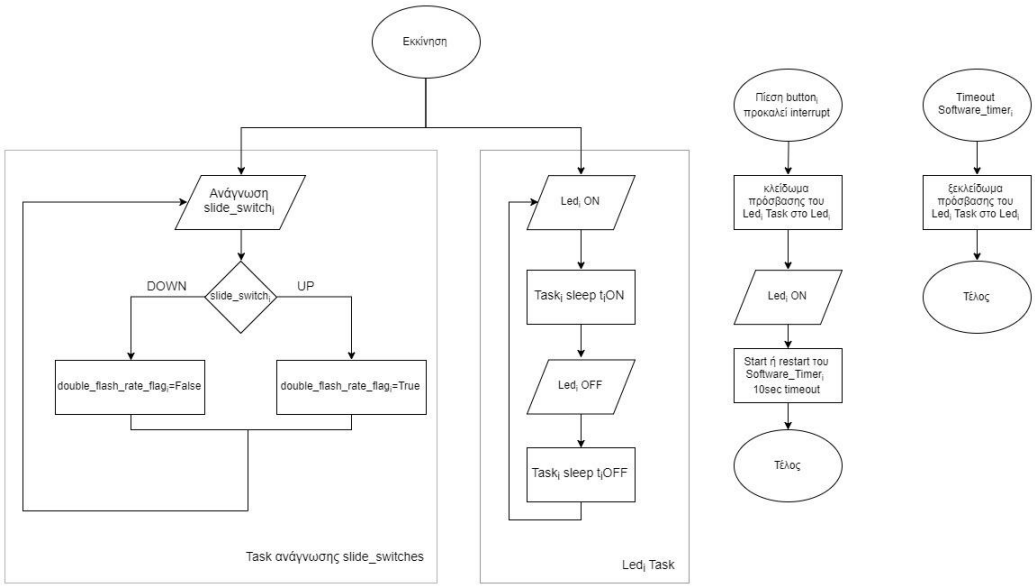
```
/*
 * See header file for description.
 */
StackType_t *pxPortInitialiseStack( StackType_t *pxTopOfStack, TaskFunction_t pxCode, void *pvPar
{
    /* Setup the initial stack of the task.  The stack is set exactly as
    expected by the portRESTORE_CONTEXT() macro.

    The fist real value on the stack is the status register, which is set for
    system mode, with interrupts enabled.  A few NULLs are added first to ensure
    GDB does not try decoding a non-existent return address. */
    *pxTopOfStack = ( StackType_t ) NULL;
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) NULL;
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) NULL;
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) portINITIAL_SPSR;

    if( ( ( uint32_t ) pxCode & portTHUMB_MODE_ADDRESS ) != 0x00UL )
    {
        /* The task will start in THUMB mode. */
        //*pxTopOfStack |= portTHUMB_MODE_BIT;
    }

    pxTopOfStack--;
```

Σχήμα 3.2.24 :Ο κώδικας του αρχείου port.c

Τώρα μπορεί να προστεθεί ο κώδικας μιας εφαρμογής που χρησιμοποιεί οποιαδήποτε από τις δυνατότητες του FreeRTOS καθώς και το GPIO (button,switches,leds) του Zybo με δυνατότητα interrupts. Μια τέτοια εφαρμογή είναι η εφαρμογή επίδειξης (Demo) που θα παρουσιαστεί στην συνέχεια.

3.3 Εφαρμογή επίδειξης του FreeRTOS στο Zybo

Πρόκειται για μια εφαρμογή στην οποία τα τέσσερα LED της πλακέτας αναβοσβήνουν με διαφορετικό ρυθμό το κάθε ένα με περιόδους 500,1000,2000 και 4000 ms. Επίσης αν κάποιος από τους τέσσερεις διακόπτες ολίσθησης είναι ενεργοποιημένος το αντίστοιχο Led αναβοσβήνει με την διπλάσια συχνότητα από αυτή που είχε πριν. Τέλος αν πατηθεί κάποιο από τα τέσσερα push buttons τότε το αντίστοιχο Led μένει αναμμένο για χρόνο ίσο 10 sec, πριν επιστρέψει στην προηγούμενη λειτουργία του. Τα παραπάνω συνοψίζονται στο διάγραμμα λειτουργίας του σχήματος 3.3.1.



Σχήμα 3.3.1 :Διάγραμμα ροής λειτουργίας της εφαρμογής επίδειξης του FreeRTOS στο Zybo.

Για κάθε LED έχουμε κι ένα FreeRTOS Task προτεραιότητας tskIDLE_PRIORITY +1. Μέσα στο loop του κάθε task γίνεται χρήση του API function του FreeRTOS ***vTaskDelayUntil(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement)*** ώστε με την χρήση της μεταβλητής LastFlashTime να αποθηκεύεται ο χρόνος (σε ticks) της εξόδου

του task από το Blocked state στο οποίο μπήκε λόγω της τελευταίας κλήσης στην ***vTaskDelayUntil(...)***. Ο χρόνος αυτός αποτελεί την βάση υπολογισμού του χρόνου της επόμενης εξόδου του Task από το Blocked state δηλαδή στο tick με αριθμό (LastFlashTime+xFlashRate/2). Η χρήση της *vTaskDelayUntil* αντί της *vTaskDelay* είναι ότι εξασφαλίζεται αυστηρός χρονισμός με προκαθορισμένη περίοδο χωρίς να επηρεάζεται από τον χρόνο εκτέλεσης του κώδικα μεταξύ δύο διαδοχικών κλήσεων στην *vTaskDelayUntil(...)*.
Επιπρόσθετα, τα LEDFlashTask έχουν την μεγαλύτερη προτεραιότητα στο σύστημα οπότε δεν θα έχουμε διακυμάνσεις(jitter) στο χρονισμό των Led, πιθανά επειδή κάποιο άλλο Task ανώτερης προταιρεότητας θα εμπόδιζε την εκτέλεση τους.

Στο σύστημα υπάρχει κι ένα ακόμα Task το οποίο διαβάζει τους διακόπτες ολίσθησης και θέτει global σημαίες (κοινή μνήμη για τα task) που υποδηλώνουν κανονική ή διπλάσια συχνότητα. Αυτό το Task έχει priority tskIDLE_PRIORITY οπότε έχουμε context switch μεταξύ αυτού και του IDLE task του FreeRTOS κάθε tick_period ms, εν προκειμένω 1ms.

```
/*-----------------------------------------------------------*/
static void vSwitchesTask(void* pvParameters ){
    uint8_t i,switch_data;
    for(;;){
     switch_data=ReadSwitches();
     for(i=0;i<ledNUMBER_OF_LEDS;i++)
        should_halve_flashrate[i]=switch_data & (1 << i);
    }
}
```

Σχήμα 3.2.2 :Ο κώδικας του Task που διαβάζει τους διακόπτες.

Τέλος η απόκριση στο πάτημα κάποιου button βασίζεται στο **interrupt** που προκαλείται,εφόσον έγινε κατάλληλος προγραμματισμός του FPGA κι παραμετροποίηση του interrupt controller του ARM core.

```
uint8_t BTN_Intr_Callback(int btn_value){
 if(btn_value) press_counter++; //if is pressed
```

Σχήμα 3.2.3 :Callback συνάρτηση καλούμενη από την ρουτίνα εξυπηρέτησης της διακοπής.

Χρησιμοποιούνται τέσσερις software timers,ένας για κάθε LED που ενεργοποιούνται μέσα από την ρουτίνα εξυπηρέτησης διακοπής κι έπειτα από τον προκαθορισμένο χρόνο εκτελείται η TimerCallback συνάρτηση που έχουμε ορίσει. Οι εντολές για παύση και επανεκκίνηση της λειτουργίας των Led βρίσκονται μέσα στην ρουτίνα εξυπηρέτησης της διακοπής και στην TimerCallback συνάρτηση. Η χρήση της FreeRTOS API συνάρτησης *xTimerReset(...)* επιτρέπει την ανανέωση του Timer αν ξαναπατηθεί το πλήκτρο ώστε να παραταθεί ο χρόνος που το αντίστοιχο Led μένει αναμμένο. Στον κώδικα χρησιμοποιείται η *xTimerResetFromISR()* που είναι ασφαλής για κλήση μέσα από ρουτίνα εξυπηρέτησης διακοπής.

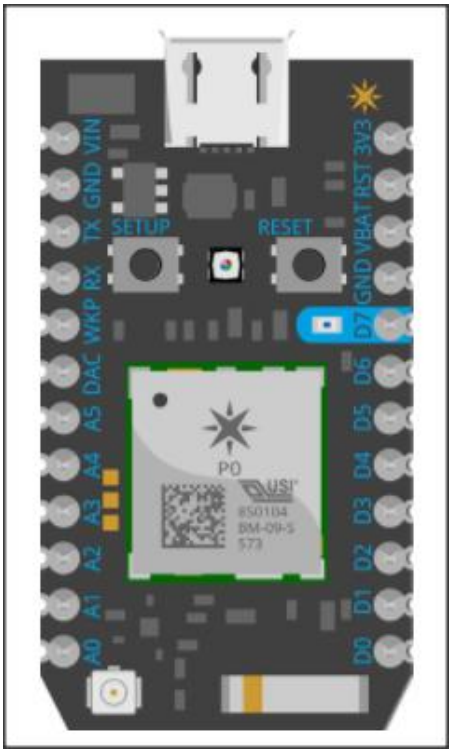
4

Εφαρμογή ανάλυσης καρδιογραφήματος ECG στην αναπτυξιακή πλακέτα Particle Photon

Η αναπτυξιακή πλακέτα Particle Photon έχει σημαντικά πλεονεκτήματα σαν αναπτυξιακό ενσωματωμένων IoT εφαρμογών. Το πολύ μικρό φυσικό μέγεθος σε συνδυασμό με την συνδεσιμότητα του μέσω Wifi καθώς και την χαμηλή κατανάλωση ενέργειας το καθιστά μια πολύ δελεαστική επιλογή για την ανάπτυξη της εφαρμογής καταγραφής κι ανάλυσης καρδιογραφήματος ΗΚΓ (ECG), καθώς ο ασθενής θα μπορεί να το έχει επάνω του σαν wearable συσκευή. Ταυτόχρονα η επεξεργαστική ισχύ του ARM Cortex M3 είναι ικανοποιητική ενώ με design space exploration μπορούν να γίνουν επιλογές που θα αφορούν ποια τμήματα της εφαρμογής θα εκτελούνται στο Photon και ποια σε κάποιο IoT gateway ή στο Cloud.

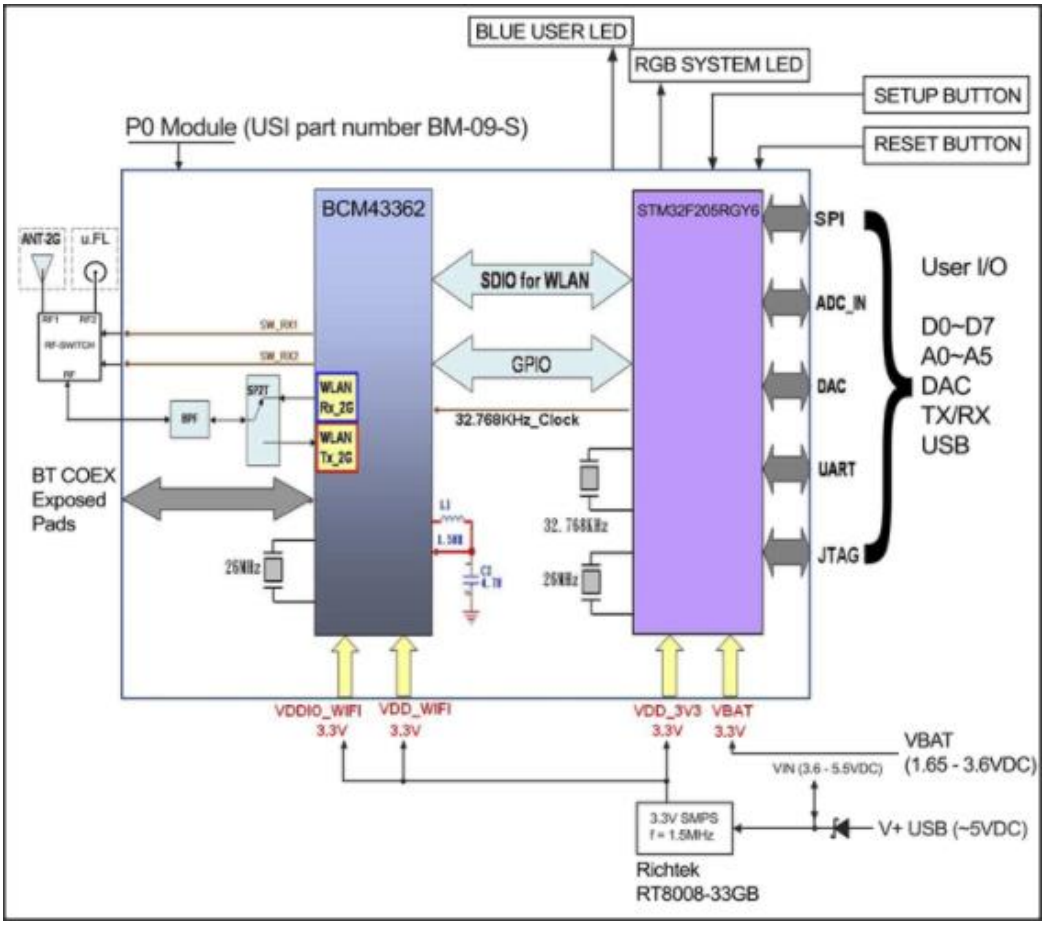
Το firmware του Photon βασίζεται στο FreeRTOS και κάνει μερικές δυνατότητες του ορατές στον προγραμματιστή όπως τους Software timers. Ωστόσο δεν υπάρχουν περιθώρια παραμετροποίησης του και γενικά ελευθερία για χρήση του FreeRTOS API, όπως κλήσεις για δημιουργία FreeRTOS tasks. Αυτός ήταν κι ο λόγος που ο βασικός πειραματισμός με το FreeRTOS έγινε σε άλλη αναπτυξιακή πλακέτα,στο Zybo όπως είδαμε.

4.1 Τεχνικά χαρακτηριστικά του Photon



Σχήμα 4.1.1 :Particle Photon

Η καρδιά του συστήματος είναι,όπως φαίνεται στην εικόνα, το P0 module το οποίο αποτελείται από το STM32F205RGY6, έναν 120Mhz ARM Cortex M3 και το Broadcom BCM43362 Wi-Fi chip. Η ομάδα ανάπτυξης λογισμικού για το Photon υλοποίησε μια διεπαφή προγραμματισμού που μοιάζει με Arduino με πολλές έτοιμες βιβλιοθήκες για τις διάφορες λειτουργίες όπως μετατροπή αναλογικό σε ψηφιακό (ADC) ή τη σύνδεση στο Particle Cloud. Στο σχήμα 4.1.2 βλέπουμε το λογικό διάγραμμα του συστήματος του Photon.

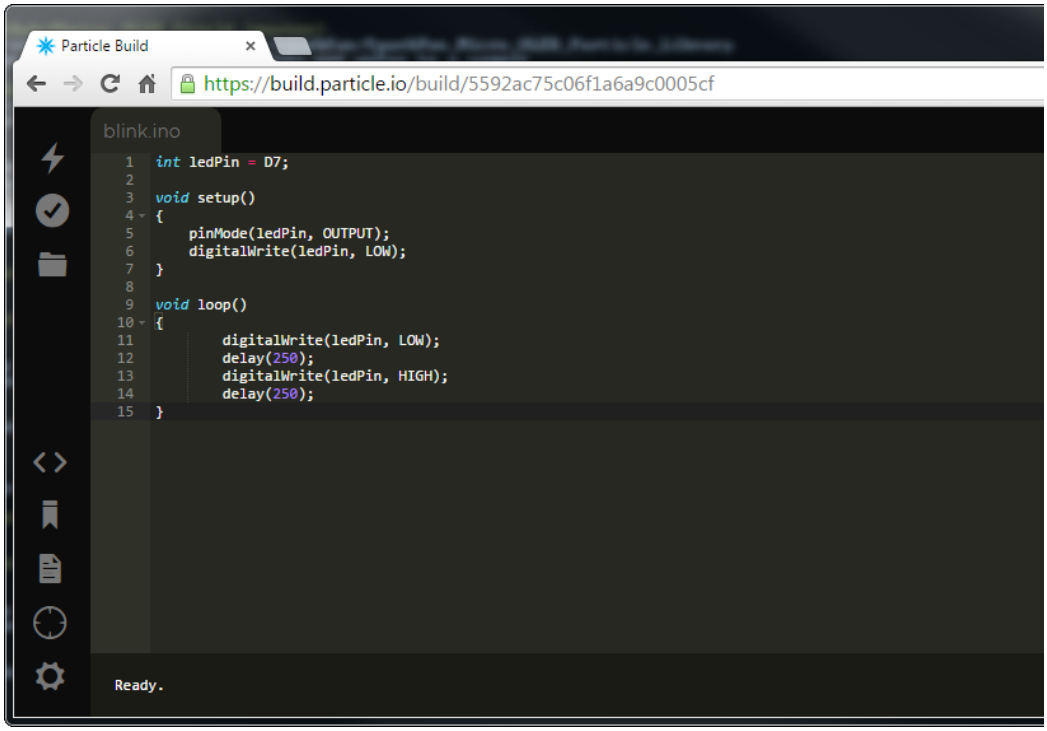


Σχήμα 4.1.2 :Block διάγραμμα του Photon[10].

Βασικά χαρακτηριστικά

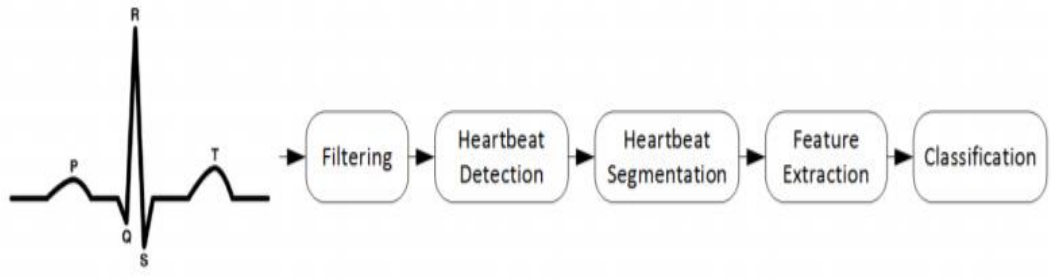
- **Processor**
  - ARM Cortex M3 120MHz
- **Memory**
  - 1MB FLASH, 128KB RAM
- **I/O**
  - 18x Digital pins
  - 8x ADC
  - 7x PWM
  - 2x DAC
  - 2x SPI
  - 1x I2C
  - 1x I2S
  - 1x CAN
  - 1x USB
  - JTAG interface pins
- **Wifi**
  - Broadcom BCM43362 IEEE 802 11b/g/n Wi-Fi chip
- **Software**
  - FreeRTOS kernel, με 2 tasks ένα με κώδικα χρήστη κι ένα για τον driver του Wifi chip.

- Arduino-like προγραμματιστική διεπαφή όπως στο σχήμα 4.1.3.



Σχήμα 4.1.3 :Online editor για το Photon.

4.2 Ανάπτυξη της εφαρμογής



Στα επόμενα περιγράφεται η πορεία της υλοποίησης της εφαρμογής ανάλυσης ECG στο Photon με αφετηρία τον πηγαίο κώδικα της εφαρμογής από προηγούμενη εργασία του εργαστηρίου[5], τις διαδοχικές τροποποιήσεις του ώστε να μπορεί να εκτελείται στο Photon μέχρι την καταγραφή κι ανάλυση πραγματικού σήματος σε πραγματικό χρόνο.

Μελέτη κώδικα εφαρμογής

Το σημείο αφετηρίας της εφαρμογής είναι ο πηγαίος κώδικας που περιλαμβάνει όλα τα στάδια ανάλυσης του καρδιογραφήματος ECG που φαίνονται στο σχήμα 4.2.1.

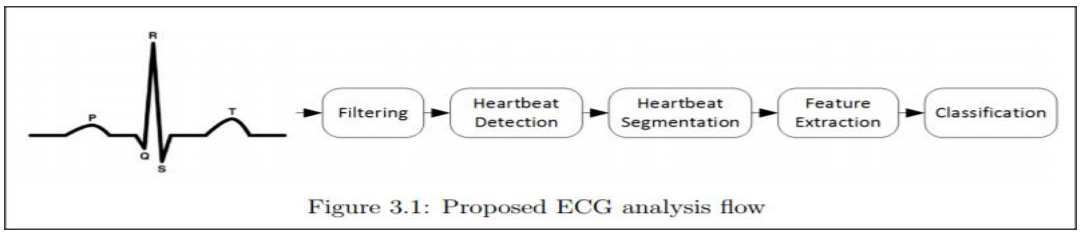
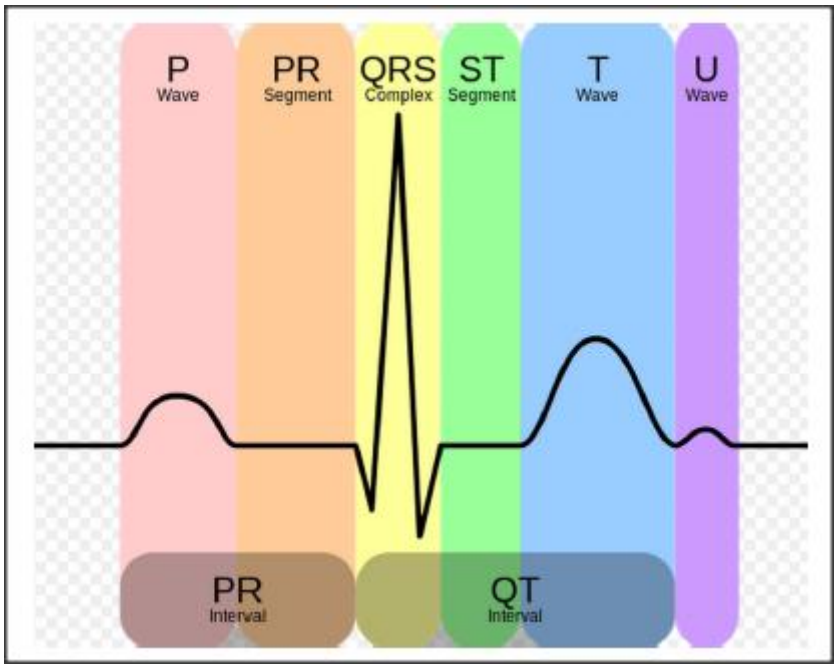


Figure 3.1: Proposed ECG analysis flow

Σχήμα 4.2.1 :Το διάγραμμα της εφαρμογής ανάλυσης[5].

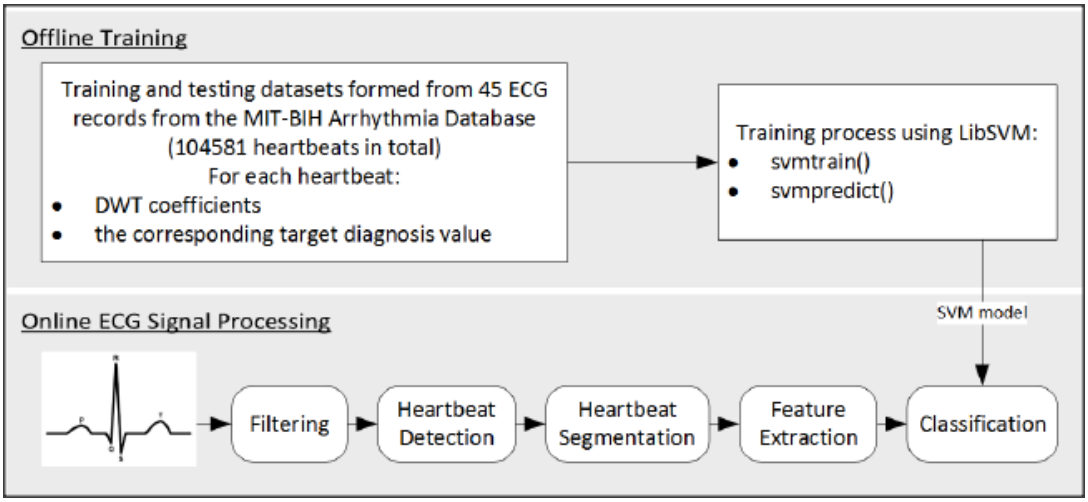
Συνοπτικά αφού ληφθεί το σήμα φιλτράρεται έτσι ώστε να αφαιρεθεί ο θόρυβος,που δεν είναι αμελητέος αφού το υπό εξέταση σήμα είναι αρκετά ασθενές. Πηγές θορύβου είναι το δίκτυο ηλεκτροδότησης (50Hz) και η δραστηριότητα των άλλων μυών του σώματος του ασθενούς όπως αυτοί της αναπνοής. Στο κώδικα υλοποιείται ένα ζωνοπερατό φίλτρο σαν συνδυασμός ενός βαθυπερατού κι ενός υψιπερατού φίλτρου με συχνότητες αποκοπής 1Hz και 50Hz. Έπειτα το παράθυρο σήματος περνάει στο στάδιο ανίχνευσης παλμών (Heartbeat Detection) ώστε να ανιχνευθούν τα QRS συμπλέγματα,σχήμα 4.2.2.



Σχήμα 4.2.2 :Μια φυσιολογική κυματομορφή καρδιογραφήματος[5].

Στο κώδικα υπάρχουν δύο επιλογές ανιχνευτών QRS: ο detector wqrs κι ο detector sqrs. Αφού εντοπιστούν οι κορυφές, το σήμα χωρίζεται σε παράθυρα εκατέρωθεν αυτών με μέγεθος παραθύρου 257 δείγματα. Έπειτα με χρήση του μετασχηματισμού dbwavelet υλοποιείται η εξαγωγή χαρακτηριστικών (feature extration) και από κάθε παράθυρο καρδιογραφήματος παράγεται ένα διάνυσμα συντελεστών (coef) μεγέθους 18. Το επόμενο και τελευταίο στάδιο αφορά τον χαρακτηρισμό του καρδιογραφήματος ως Φυσιολογικό(Normal) ή Ανώμαλο (Abnormal) με χρήση Support Vector Machine το οποίο λαμβάνει ως είσοδο το προηγούμενο διάνυσμα συντελεστών. Η εκπαίδευση του αλγορίθμου έχει προηγηθεί (offline) και υλοποιήθηκε με βάση δύο σύνολα δεδομένων

(καρδιογραφημάτων) για τα οποία είναι ο γνωστός ο χαρακτηρισμός τους από την MIT-BIH Arrhythmia Database[11].Το ένα σύνολο είναι το σύνολο εκμάθησης (training set) και το δεύτερο είναι το σύνολο ελέγχου (test set) και με βάση αυτά, με επαναληπτικές δοκιμές, επιλέγονται τα support vectors του SVM που δίνουν τα καλύτερα αποτελέσματα.Το σχήμα 4.2.3 δείχνει το σύνολο της εφαρμογής.



Σχήμα 4.2.3 :Το πλήρες διάγραμμα λογισμικού της εφαρμογής[5].

Όπως αναφέρθηκε ο αρχικός κώδικας προορίζεται για εκτέλεση σε περιβάλλον Linux γεγονός που δημιουργεί προβλήματα στην εκτέλεση του στο Photon.Αυτά είναι:

- Απουσία συστήματος αρχείων στο Photon.
- Περιορισμένη μνήμη του Photon.
- Απουσία standard input output.
- Μέγεθος στοίβας συναρτήσεων και δυναμική διαχείριση μνήμης.

Απαίτηση συστήματος αρχείων

Στον κώδικα πολλές παράμετροι για τα διάφορα στάδια του ECG flow διαβάζονταν από αρχεία όπως φαίνεται παρακάτω:

```
fp = fopen(argv[1],"r");    //open file containing the feature vector configuration
fgets(line,80,fp);
if (strcmp(line,"coefficients\n")) printf("Error in config file.\n");
fgets(line,80,fp);
ac1 = atoi(line);
```

Σχήμα 4.2.4 :Διάβασμα από αρχείο παραμετροποίησης.

Η λύση ήταν τα απαραίτητα δεδομένα να αποθηκευτούν σαν C πίνακες σε header αρχεία, hard-coded δηλαδή στον κώδικα της εφαρμογής. Συγκεκριμένα,η απόκριση του φίλτρου,τα dbwavelets καθώς και δύο παράθυρα 512 δειγμάτων ECG σήματος για την επαλήθευση της ορθής λειτουργίας ενσωματώθηκαν με αυτόν τον τρόπο στον κώδικα της εφαρμογής.

Περιορισμένη μνήμη

Το Photon διαθέτει 1MB Flash μνήμη στην οποία,ως γνωστόν,αποθηκεύονται ο κώδικας της εφαρμογής καθώς και η αρχικοποιημένες μεταβλητές. Τέτοιες είναι όλα τα παραπάνω δεδομένα που έγιναν hard-coded γεγονός που περιορίζει ακόμα περισσότερο το διαθέσιμο χώρο. Αυτός ο περιορισμός οδήγησε στο να μην χρησιμοποιηθεί το αρχικό svm_model αλλά η βασική μορφή ενός support vector machine με δύο πίνακες support_vectors και coefficients μεγέθους 1274.Επιπλέον,για το φιλτράρισμα ενώ αρχικά χρησιμοποιούνταν ένα βαθυπερατό

κι ένα υψιπερατό φίλτρο,στο Photon τελικά χρησιμοποιήθηκε μόνο το βαθυπερατό φίλτρο οπότε εξοικονομήθηκε ο χώρος του πίνακα απόκρισης του υψιπερατού φίλτρου.Ο συμβιβασμός αυτός επιλέχθηκε καθώς το σημαντικότερο μέρος του θορύβου αποκόπτεται από το βαθυπερατό φίλτρο.

Απουσία standard input output

Στον αρχικό κώδικα χρησιμοποιούνταν η printf για την εμφάνιση μηνυμάτων στο τερματικό για την επίβλεψη της ορθής λειτουργίας του προγράμματος καθώς και για debugging. Στο Photon σαν έξοδος μηνυμάτων χρησιμοποιήθηκε η σειριακή θύρα.

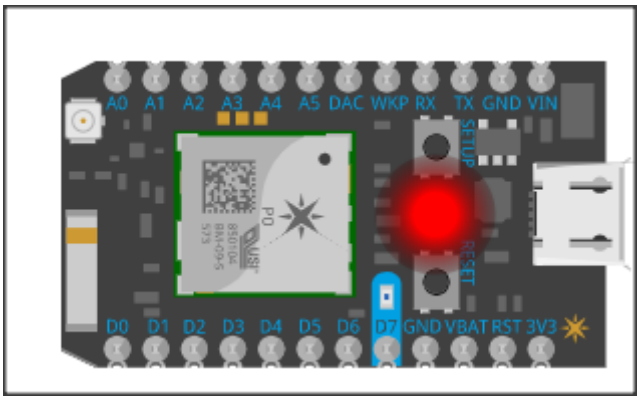
```
Serial.printf("ac1=%d,ac2=%d,ac3=%d,ac4=%d,dc1=%d,dc2=%d,dc3=%d,dc4=%d,fv_size is %
```

Σχήμα 4.2.5 :Παράδειγμα εκτύπωσης μηνύματων με πληροφορίες παραμετροποίησης της εφαρμογής όπως το μέγεθος του feature vector.

Συνεπώς όλες η printf μετατράπηκαν σε Serial.printf,όπως στο σχήμα 4.2.5, κι επίσης προστέθηκαν νέες για debugging στα διάφορα σημεία της εφαρμογής.

Μέγεθος στοίβας συναρτήσεων και δυναμική διαχείριση μνήμης

Μετά την εφαρμογή όλων των παραπάνω τροποποίησεων δημιουργήθηκε ένα πρώτο port για το Photon. Ωστόσο η εκτέλεση διακόπτονταν λόγω stack-overflow όπως πληροφορούσε το ενδεικτικό Led στην πλακέτα,σχήμα 4.2.6.



Σχήμα 4.2.6 :Photon SOS blink.

Το πρόβλημα προκαλούσε το γεγονός ότι οι πίνακες που χρησιμοποιούσαν κάποιες C συναρτήσεις μόνο για ανάγνωση δηλώνονταν μέσα στο σώμα της συνάρτησης οπότε δεσμευόταν χώρος στοίβας της συνάρτησης που έχει άνω όριο. Έτσι αυτοί οι πίνακες δηλώθηκαν ως global οπότε ο linker δεσμεύει εκ των προτέρων τον απαραίτητο χώρο. Τέλος για αξιοπιστία, έγιναν όλες οι malloc ασφαλείς με έλεγχο για αποτέλεσμα null.

Μετά από τα παραπάνω η εφαρμογή άρχισε να εκτελείται κανονικά στο Photon και να δίνει τα αναμενόμενα αποτελέσματα.Στη συνέχεια,προστέθηκε και χρονομέτρηση του κάθε σταδίου του ECG flow:

```
t0=millis();
  sig = noiseremoval(sig,len);


elapsed=millis()-t0;
Serial.printf("Noiseremoval:%ld ms\n",elapsed);
```

Σχήμα 4.2.7 :Χρήση της millis() για χρονομέτρηση του σταδίου φιλτραρίσματος.

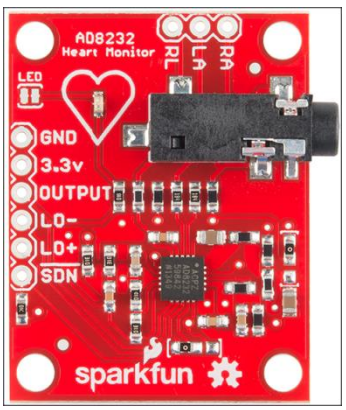
Η millis() είναι μια συνάρτηση που παρέχει το firmware του Photon κι επιστρέφει το χρόνο σε millisecond από την στιγμή που ξεκίνησε η εκτέλεση του κώδικα.Η ακρίβεια της είναι πολύ καλή καθώς βασίζεται σε interrupts που παράγει ένας hardware timer του μικροεπεξεργαστή. Παρακάτω φαίνονται τα αποτελέσματα της εκτέλεσης και της χρονομέτρησης όπως τυπώνονται στο σειριακό τερματικό.

```
Noiseremoval:347 ms
Selected detector:0 ms
Dbwavelet:6 ms
Classification:41 ms
rpeaks: 1
                        normal: 1
                        abnormal: 0
Noiseremoval:348 ms
Selected detector:1 ms
Dbwavelet:6 ms
Classification:41 ms
rpeaks: 1
                        normal: 1
                        abnormal: 0
Noiseremoval:348 ms
Selected detector:0 ms
Dbwavelet:6 ms
Classification:41 ms
rpeaks: 1
                        normal: 1
                        abnormal: 0
Noiseremoval:348 ms
Selected detector:1 ms
Dbwavelet:6 ms
Classification:41 ms
rpeaks: 1
```

Σχήμα 4.2.8 :Χρονομέτρηση κι αποτελέσματα του ECG flow στο Photon.

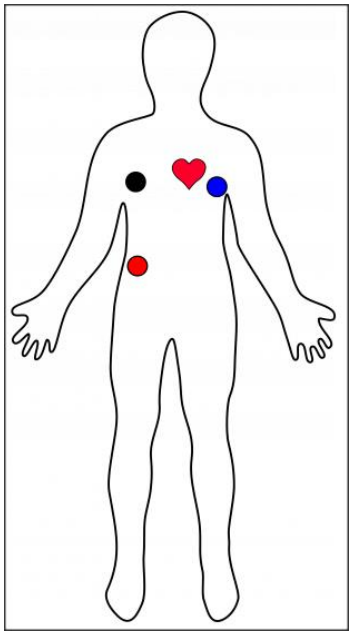
<u>Καταγραφή πραγματικού σήματος κι ανάλυση σε πραγματικό χρόνο</u>

Το επόμενο βήμα ήταν η προσθήκη της δυνατότητας καταγραφής καρδιογραφήματος από ασθενή σε πραγματικό χρόνο.Για το σκοπό αυτό χρησιμοποιήθηκε το chip AD8232 στην παρακάτω πλακέτα:



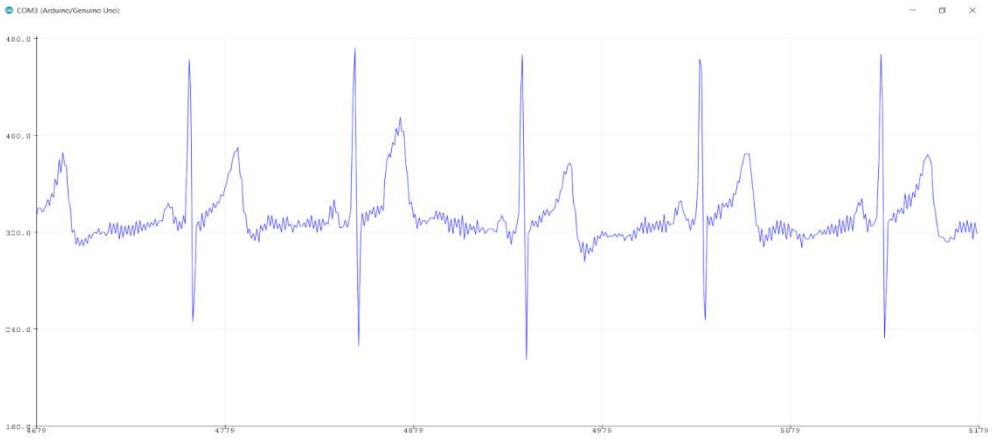
Σχήμα 4.2.9 :SparkFun Single Lead Heart Rate Monitor-AD8232[12].

Το παραπάνω συνδέεται με 3 αγωγούς στο σώμα του ασθενούς στα σημεία που φαίνονται παρακάτω:



Σχήμα 4.2.10 :Σημεία λήψης καρδιογραφήματος.

Το σήμα του ΗΚΓ είναι διαθέσιμο σαν αναλογική τάση 0-3.3V στον ακροδέκτη εξόδου κι έτσι με έναν Analog to Digital Converter(ADC) μπορούμε να λάβουμε το ψηφιακό σήμα για επεξεργασία. Παρακάτω φαίνεται ένα στιγμιότυπο από την απεικόνιση καρδιογραφήματος εθελοντή σε πραγματικό χρόνο με το AD8232:



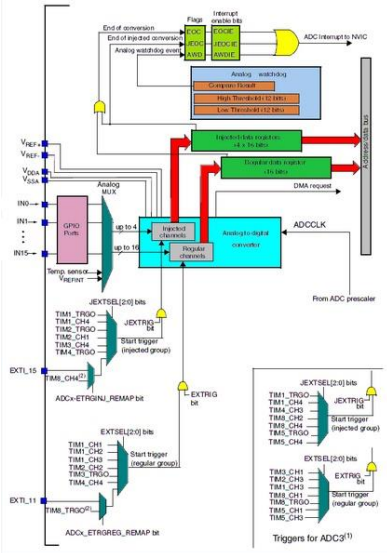
Σχήμα 4.2.11 :Απεικόνιση καρδιογραφήματος που λαμβάνεται από Η/Υ με σειριακή επικοινωνία.

Για την παραπάνω επιβεβαίωση της ορθής λειτουργίας του AD8232 χρησιμοποιήθηκε ένα Arduino Uno(AVR AtMega328) που υλοποίησε την δειγματοληψία 330Hz 10-bit και την αποστολή των δεδομένων στον Η/Υ για απεικόνιση.

Στο Photon για την δειγματοληψία και μετέπειτα ανάλυση του σήματος η έτοιμη συνάρτηση analogRead(Pin) (12bit), η οποία παρέχεται από το firmware, δεν είναι κατάλληλη διότι η υλοποίηση της είναι blocking, που σημαίνει ότι ο επεξεργαστής κάνει polling μια σημαία που δηλώνει ολοκλήρωση της μετατροπής από αναλογικό σε ψηφιακό. Συνεπώς αν την

χρησιμοποιούσαμε θα έπρεπε για ένα χρονικό διάστημα το Photon να δειγματοληπτεί και να αποθηκεύει το σήμα κι ένα επόμενο χρονικό διάστημα να εκτελεί το ECG flow για την ανάλυση. Αυτό θα σήμαινε πως πιθανές ανωμαλίες του καρδιογραφήματος θα περνούσαν απαρατήρητες αν συνέβαιναν στο διάστημα της μη δειγματοληψίας. Υπάρχει, δηλαδή, η απαίτηση για παράλληλη καταγραφή την ώρα της ανάλυσης του ECG σήματος έτσι ώστε να υπάρχει συνεχής ροή δεδομένων κι ανάλυση ολόκληρου του σήματος και όχι μόνο κάποιων χρονικών παραθύρων του.

Τη λύση σε αυτό το πρόβλημα έδωσαν οι δυνατότητες που παρέχει η συγκεκριμένη σειρά MCU STM32F205RGY6 για DMA (Direct Memory Access) της μονάδας ADC καθώς κι ενεργοποίηση της με εξωτερικό σκανδαλισμό (external trigger) με χρήση hardware timer. Επομένως μπορούμε να επιλέξουμε την επιθυμητή συχνότητα δειγματοληψίας ρυθμίζοντας την περίοδο του timer που θα ενεργοποιεί την μονάδα ADC η οποία θα γράφει τα δεδομένα κατευθείαν στην μνήμη χωρίς απασχόληση του επεξεργαστή.



Σχήμα 4.2.12 :Το ADC block του STM32F205RGY6.

Όπως φαίνεται κι από την παραπάνω εικόνα το ADC block δίνει πάρα πολλές επιλογές παραμετροποίησης γεγονός που δυσχεραίνει την εύρεση της σωστής. Τελικά,επιλέχθηκαν οι κατάλληλες ρυθμίσεις των καταχωρητών ελέγχου (hardware registers) ώστε ο μετρητής (hardware timer) TIM3 να σκανδαλίζει(trigger) την μονάδα ADC με συχνότητα 360Hz (συχνότητα δειγματοληψίας της βάσης MIT-BIH Arrhythmia Database) κι έπειτα από κάθε μετατροπή τα αποτελέσματα να γράφονται σε μια περιοχή μνήμης (buffer) της RAM. Ο buffer αυτός έχει το διπλάσιο μέγεθος από το παράθυρο σήματος που λαμβάνει η Flow για ανάλυση και χαρακτηρισμό. Επομένως όταν γίνεται ανάλυση ενός παραθύρου καταγεγραμμένου σήματος παράλληλα διαβάζεται το επόμενο του κι αποθηκεύεται στο άλλο μισό του buffer ώστε να διασφαλίζεται η αδιάκοπη ροή δεδομένων.

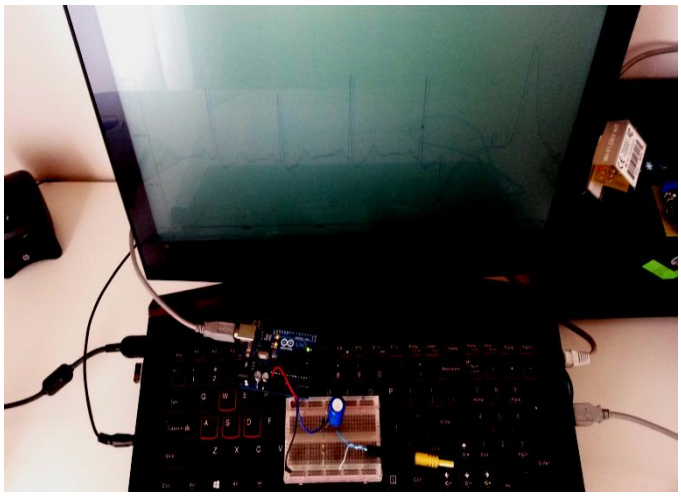
Προφανώς όλη η παραπάνω ανάπτυξη της εφαρμογής απαιτούσε κάποιον εθελοντή να είναι διαρκώς συνδεδεμένος και να παρέχει το καρδιογράφημα του, κάτι που αποδείχθηκε διόλου πρακτικό. Προέκυψε δηλαδή η ανάγκη προσομοίωσης, παραγωγής δηλαδή αναλογικού σήματος καρδιογραφήματος που θα δίνεται ως είσοδος στο Photon. Η απλούστερη λύση που βρέθηκε ήταν η χρήση της κάρτας ήχου ενός υπολογιστή κι η αναπαραγωγή καρδιογραφήματος σαν ηχητικό (αναλογικό) σήμα. Επιλέχθηκε το αρχείο σήματος data_101 της MIT-BIH Database:

Reference annotations	Signals	Header
100.atr	100.dat	100.hea
101.atr	101.dat	101.hea

Σχήμα 4.2.13 :Καρδιογράφημα ασθενή 101 της MIT-BIH Database.

Με χρήση Matlab έγινε upsampling από τα 360Hz στα 8000Hz (ελάχιστο sampling rate για την μορφή ήχου .wav) καθώς και αφαίρεση dc συνιστώσας,κανονικοποίηση και μετατροπή σε .wav.

Στην αναλογική είσοδο A0 του Photon συνδέουμε έναν διαιρέτη τάσης ώστε να πολωθεί στα 3.3V/2=1.5V και συνδέουμε το ένα κανάλι ήχου με χρήση ενός πυκνωτή(ac coupling). Με αυτόν τον τρόπο εξασφαλίζουμε ότι δεν θα υπάρχει ψαλιδισμός για σήμα <0V κι επομένως στο Photon λαμβάνουμε το σήμα γύρω από την τιμή 2048(3.3V->4096 12bit ADC).



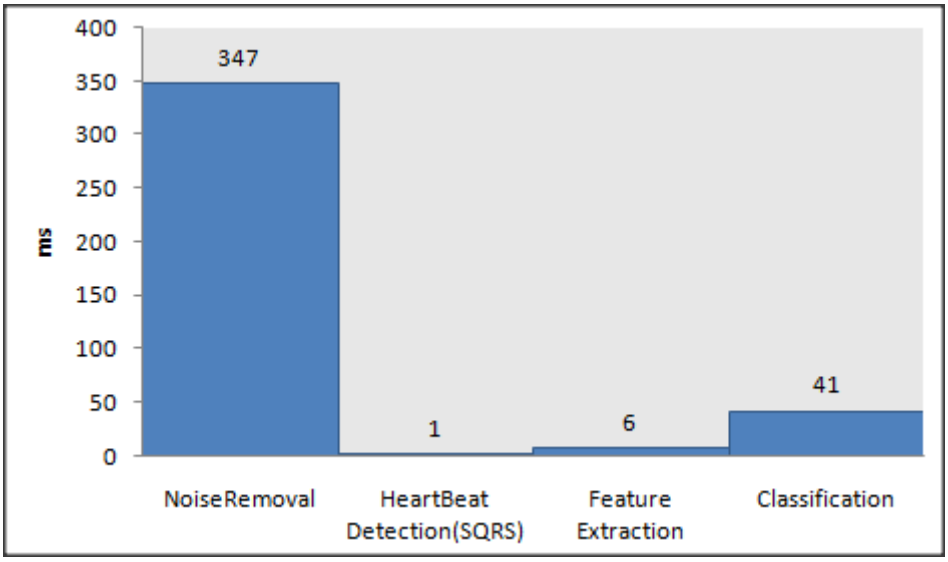
Σχήμα 4.2.14 :Επαλήθευση της αναπαραγωγής καρδιογραφήματος από την κάρτα ήχου με Arduino UNO.

Έπειτα στο Photon προστέθηκε η δυνατότητα αποστολής των παραθύρων σήματος που καταγράφει καθώς και το αποτέλεσμα της ανάλυσης, Φυσιολογικό(Normal) ή Ανώμαλο(Abnormal), συμπεριλαμβανομένου και του πίνακα coef που προέκυψε από το feature extraction με dbwavelet μετασχηματισμού, για καλύτερη εποπτεία της εφαρμογής. Αυτά γράφονται σε ένα buffer κι αποστέλλονται από το Photon, μέσω Wifi, σαν μηνύματα UDP σε οποιοδήποτε απομακρυσμένο σταθμό.Κατά αυτό τον τρόπο είναι δυνατή η εποπτεία της καρδιακής λειτουργίας του ασθενούς όπου κι αν βρίσκεται.

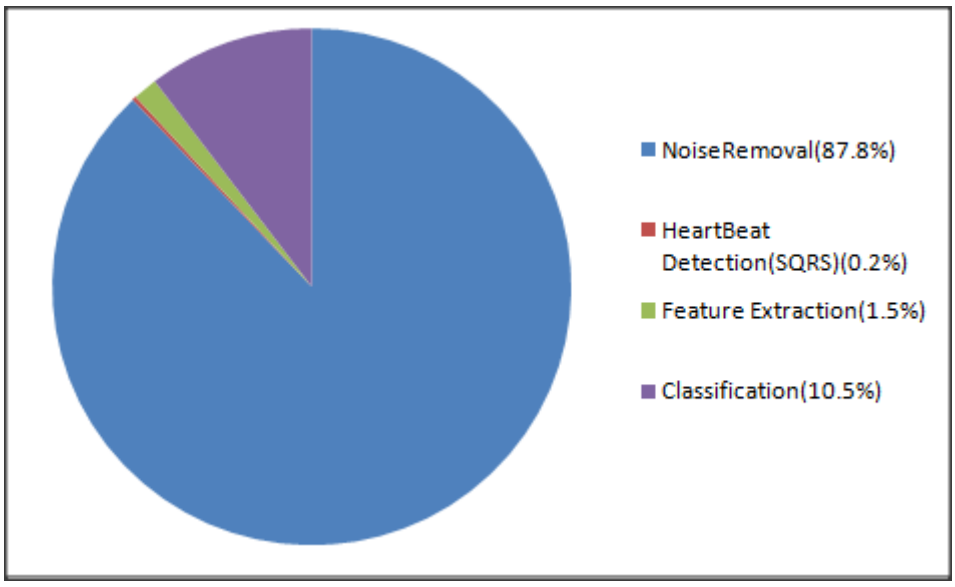
Για την επιβεβαίωση της ορθής λειτουργίας όλων των παραπάνω γράφηκε ένα απλό scipt σε Python ώστε ένας Η/Υ να λαμβάνει τα μηνύματα UDP από το Photon να τυπώνει τα δεδομένα στο terminal και να τα γράφει σε ένα αρχείο για μετέπειτα αξιολόγηση.

4.3 Πειραματικά αποτελέσματα

Η βασική παράμετρος που μελετήθηκε μετά την ανάπτυξη της ECG flow εφαρμογής στο Particle Photon ήταν ο χρόνος εκτέλεσης του κάθε σταδίου της. Βασιζόμενοι στο χρήση της συνάρτησης millis(), όπως είδαμε πριν, προέκυψαν οι μέσοι χρόνοι που απεικονίζονται στο σχήμα 4.3.1 και στο σχήμα 4.3.2 σε ποσοστιαία μορφή.

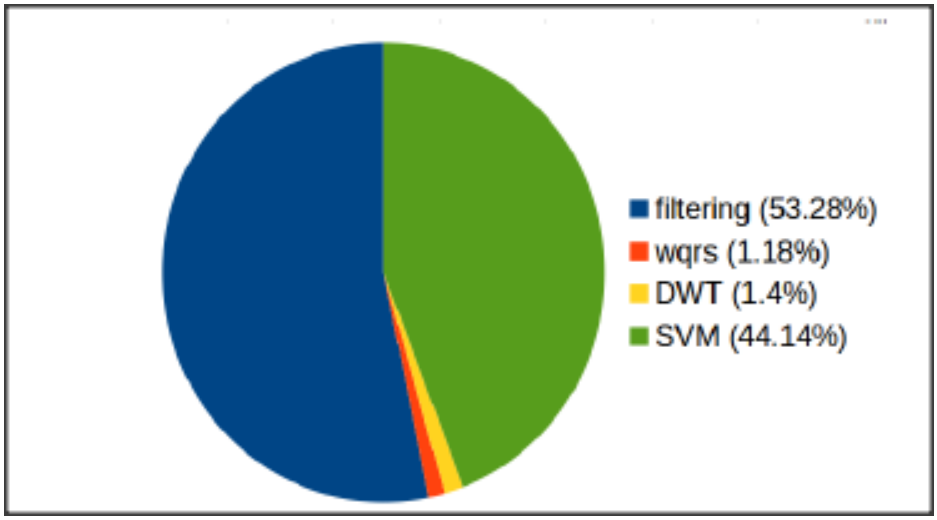


Σχήμα 4.3.1 :Χρονομέτρηση της εκτέλεσης της εφαρμογής στο Particle Photon.



Σχήμα 4.3.2 :Ποσοστό χρόνου εκτέλεσης κάθε σταδίου επί του συνολικού χρόνου εκτέλεσης της εφαρμογής στο Particle Photon.

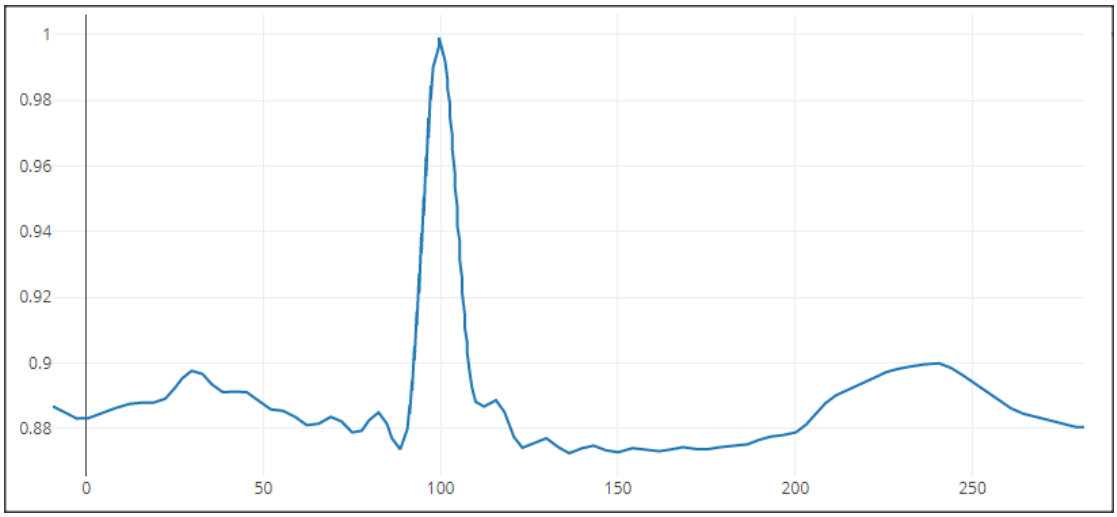
Συγκρίνοντας με το παρακάτω αποτελέσμα, σχήμα 4.3.3, της ανάλυσης στο Intel Galileo[5] :



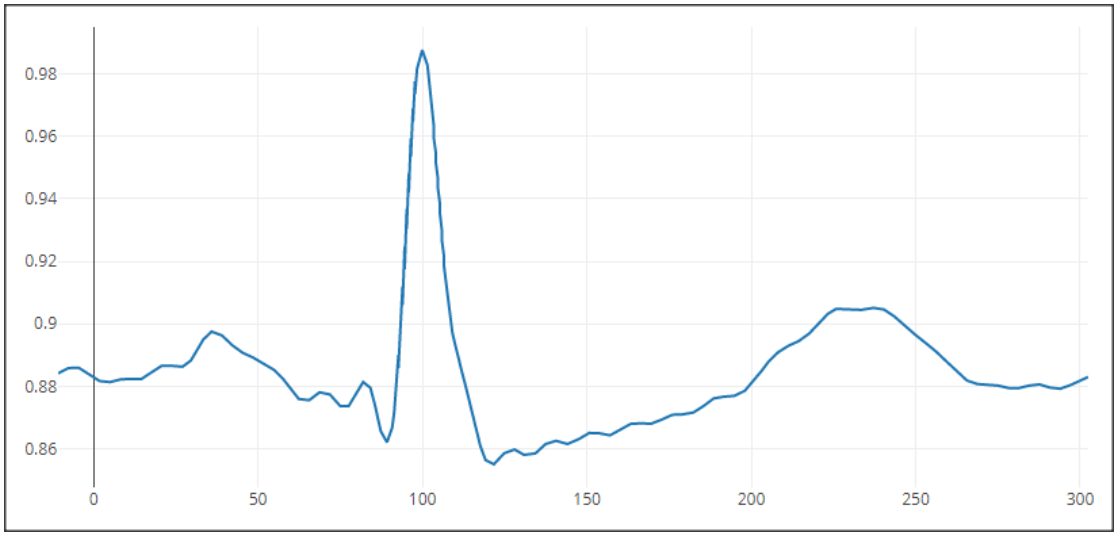
Σχήμα 4.3.3 : Ποσοστό χρόνου εκτέλεσης κάθε σταδίου επί του συνολικού χρόνου εκτέλεσης της εφαρμογής στο Intel Galileo[5].

Βλέπουμε κι εδώ ότι το μεγαλύτερο μέρος του χρόνου εκτέλεσης καταλαμβάνει το στάδιο του φιλτραρίσματος του σήματος. Ταυτόχρονα παρατηρούμε και μεγάλο ποσοστό του χρόνου για το classification κάτι το οποίο δεν παρατηρούμε στο διάγραμμα της εφαρμογής στο Photon. Αυτό δικαιολογείται από την επιλογή ενός μικρότερου support_vector[18][1274] λόγω περιορισμού της μνήμης του Photon διατηρώντας ωστόσο ικανοποιητική ακρίβεια.

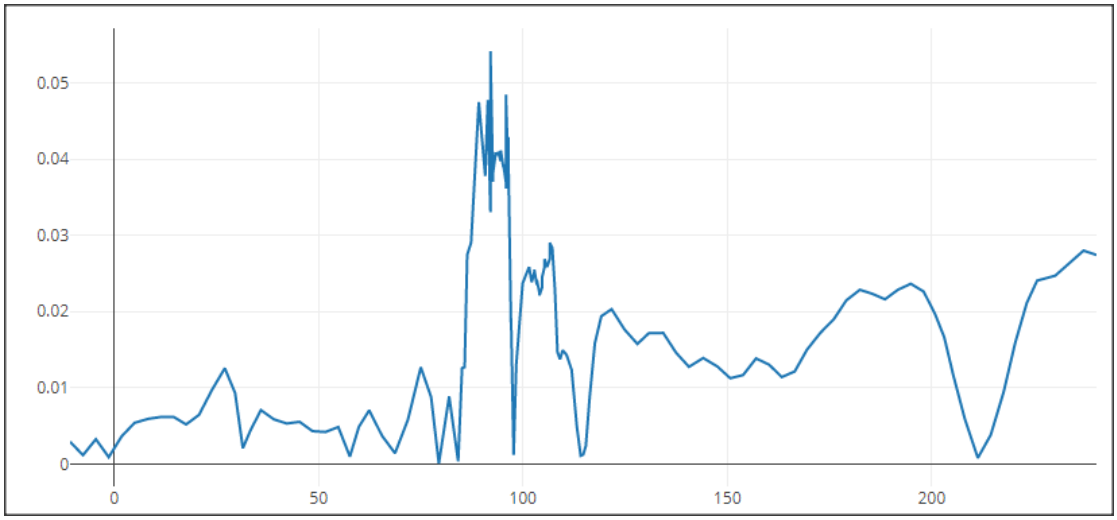
Στο επόμενο τμήμα που αποτελεί κι επέκταση της εφαρμογής αξιολογήθηκε η δυνατότητα καταγραφής πραγματικού αναλογικού σήματος από το Photon κι η αποστολή του μέσω Wifi σε Η/Υ:



Σχήμα 4.3.4 :Ένα παράθυρο σήματος γύρω από μια κορυφή στο αρχείο data_101 της MIT-BIH database(κανονικοποιημένο).



Σχήμα 4.3.5 :Το ίδιο παράθυρο σήματος όπως το διάβασε το Photon από την αναλογική του είσοδο και το απέστειλε μέσω Wifi σε Η/Υ(κανονικοποιημένο).



Σχήμα 4.3.6 :Το διάγραμμα των διαφορών του αρχικού σήματος κι αυτού που απέστειλε το Photon.

Το σχήμα 4.3.6 απεικονίζει τις διαφορές μεταξύ του αρχικού σήματος, σχήμα 4.3.4, και του σήματος που απέστειλε το Photon μέσω Wifi,σχήμα 4.3.5, αφού το διάβασε από την αναλογική του είσοδο την οποία λάμβανε από την κάρτα ήχου του υπολογιστή που εξομοίωνε το καρδιογράφημα ασθενούς. Τα σφάλματα όπως βλέπουμε είναι της τάξης του 5% και οφείλονται κατά κύριο λόγο στο τρόπο αναπαραγωγής του καρδιογραφήματος.

5 *Επίλογος*

5.1 Σύνοψη και συμπεράσματα

Η εφαρμογή ανάλυσης καρδιογραφήματος ECG αποτελείται,όπως είδαμε, από τις εξής επιμέρους λειτουργίες: καταγραφή του αναλογικού σήματος, ανάλυση του καρδιογραφήματος και σύνδεση στο διαδίκτυο μέσω Wifi,ταυτόχρονα υπάρχει η απαίτηση η εφαρμογή να είναι ντετερμινιστική και πραγματικού χρόνου. Επιπλέον υπάρχουν περιορισμοί στους διαθέσιμους πόρους μνήμης και υπολογιστικής ισχύος καθώς ο κόμβος πρέπει είναι φθηνός, μικρός σε μέγεθος και σε κατανάλωση ισχύος. Τα παραπάνω κάνουν την χρήση του FreeRTOS μια πολύ δελεαστική επιλογή καθώς με τη χρήση του μπορούν εύκολα να ικανοποιηθούν οι παραπάνω απαιτήσεις. Δηλαδή με ένα FreeRTOS task για την καταγραφή του σήματος καρδιογραφήματος, ένα για την ανάλυση του, ένα για την αποστολή των δεδομένων στο διαδίκτυο κι ένα για το Wifi chip driver –TCP/IP stack μπορεί να υλοποιηθεί η παραπάνω εφαρμογή με χρήση του FreeRTOS.

5.2 Μελλοντικές επεκτάσεις

Η υλοποίηση της εφαρμογής καταγραφής κι ανάλυσης καρδιογραφήματος ECG με χρήση του FreeRTOS ,σε έναν ARM-Cortex M3 για παράδειγμα, αποτελεί μια μελλοντική επέκταση αυτής της εργασίας σε συνδυασμό με μελέτη της ακρίβειας και της αξιοπιστίας του συστήματος. Όσα παρουσιάστηκαν εδώ αποτελούν το υπόβαθρο για αυτό το βήμα, ενώ παράλληλα η ευλιγισία που παρέχει το FreeRTOS κάνει εύκολη την προσθήκη και νέων δυνατοτήτων.

6 *Βιβλιογραφία*

[1] BORGOHAIN, Tuhin; KUMAR, Uday; SANYAL, Sugata. Survey of Operating Systems for the IoT Environment. arXiv preprint arXiv:1504.02517, 2015.

[2] MILINKOVIĆ, Aleksandar; MILINKOVIĆ, Stevan; LAZIĆ, Ljubomir. Choosing the right RTOS for IoT platform. 2015.

[3] GRAHAM, Bill; WEINSTEIN, Michael. The RTOS as the engine powering the internet of things. *white paper*, 2014.

[4] SAMIE, Farzad; BAUER, Lars; HENKEL, Jörg. IoT technologies for embedded computing: A survey. In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*. IEEE, 2016. p. 1-10.

[5] Δημητρα, Αζαριαδη. "Software Design And Optimization Of Ecg Signal Analysis And Diagnosis For Embedded Iot Devices." ,2016.

[6] BARRY, Richard. Mastering the FreeRTOS Real Time Kernel-a Hands On Tutorial Guide. *Real Time Engineers Ltd*, 2016.

[7] ZYNQ, Xilinx. 7000. *Zynq-7000 all programmable soc overview, advance product specification-ds190 (v1. 2) available on: http://www. xilinx. com/support/documentation-/data sheets/-ds190-Zynq-7000-Overview. pdf," August*, 2012.

[8] DIGILENT, Z. Y. B. O. Reference Manual, ZYBO rev. 2014.

[9] Vivado Design Suite-HLx Editions, Xilinx. *available on https://www.xilinx.com/products/design-tools/vivado.html" January*, 2018.

[10] Particle Photon,Particle. *available on https://docs.particle.io/guide/getting-started/intro/photon/" January* , 2018.

[11] MARK, R.; MOODY, G. MIT-BIH arrhythmia database directory. *Cambridge: Massachusetts Institute of Technology*, 1988.

[12] AD8232 Heart Rate Monitor,Adafruit. *available on https://* https://learn.sparkfun.com/tutorials/ad8232-heart-rate-monitor-hookup-guide*" January* , 2018.