ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Σύστημα μετρήσεων με τουλάχιστον 3 αναλογικά κανάλια μεγάλης ακρίβειας και ανάπτυξη προγράμματος σε Η/Υ για την επεξεργασία των μετρήσεων»

Του φοιτητή
Γκίκα Ανδρέα
Αρ. Μητρώου: 515018

Επιβλέπων
Άγγελος Γιακουμής
Αναπληρωτής καθηγητή

Ημερομηνία 5-6-2026

Τίτλος Δ.Ε. Σύστημα μετρήσεων με τουλάχιστον 3 αναλογικά κανάλια μεγάλης ακρίβειας και ανάπτυξη προγράμματος σε Η/Υ για την επεξεργασία των μετρήσεων.

Κωδικός Δ.Ε. 21373

Ονοματεπώνυμο φοιτητή Ανδρέας Γκίκας

Ονοματεπώνυμο εισηγητή Άγγελος Γιακουμής

Ημερομηνία ανάληψης Δ.Ε. 28-01-2022

Ημερομηνία περάτωσης Δ.Ε. 29-05-2026

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Στην Άρτεμη και στον Έκτορα

Πρόλογος

Η παρούσα διπλωματική εργασία εκπονήθηκε στο πλαίσιο των σπουδών μου με σκοπό τη μελέτη και υλοποίηση ενός ολοκληρωμένου συστήματος συλλογής δεδομένων υψηλής ταχύτητας. Η επιλογή του συγκεκριμένου θέματος έγινε λόγω του ιδιαίτερου ενδιαφέροντός μου για τα ενσωματωμένα συστήματα, τα ηλεκτρονικά συστήματα μετρήσεων και την ανάπτυξη λογισμικού χαμηλού επιπέδου.

Κατά τη διάρκεια της εργασίας είχα την ευκαιρία να ασχοληθώ με διαφορετικούς τομείς της ηλεκτρονικής και της πληροφορικής, όπως η λειτουργία μετατροπέων αναλογικού σε ψηφιακό σήμα, η αρχιτεκτονική μικροελεγκτών ARM Cortex-M7, οι μηχανισμοί DMA, τα πρωτόκολλα επικοινωνίας USB και η ανάπτυξη εφαρμογών σε Python. Παράλληλα, απέκτησα εμπειρία στην ανάλυση τεχνικών προβλημάτων και στη σχεδίαση ολοκληρωμένων συστημάτων που συνδυάζουν υλικό και λογισμικό.

Η υλοποίηση του συστήματος απαίτησε τη μελέτη και την κατανόηση τεχνολογιών που χρησιμοποιούνται ευρέως στη βιομηχανία και στην έρευνα, προσφέροντας πολύτιμη πρακτική εμπειρία πέρα από το θεωρητικό υπόβαθρο των σπουδών μου. Θεωρώ ότι η ενασχόληση με το συγκεκριμένο αντικείμενο συνέβαλε σημαντικά στην ανάπτυξη των τεχνικών μου γνώσεων και δεξιοτήτων.

Φυσικά και η ίδια η σύνταξη μιας επιστημονικής εργασίας ήταν μια σημαντική εμπειρία, που αποτελεί εφόδιο για την επιστημονική και επαγγελματική μου εξέλιξη.

Περίληψη

Αντικείμενο της παρούσας διπλωματικής εργασίας είναι η σχεδίαση και υλοποίηση ενός συστήματος συλλογής δεδομένων υψηλής ταχύτητας με τρία ανεξάρτητα αναλογικά κανάλια μέτρησης και η ανάπτυξη λογισμικού υπολογιστή για τη λήψη και απεικόνιση των μετρήσεων.

Το σύστημα βασίζεται στον μικροελεγκτή STM32H743ZI με πυρήνα ARM Cortex-M7 και αξιοποιεί τους τρεις ενσωματωμένους μετατροπείς αναλογικού σε ψηφιακό σήμα (ADC). Οι ADC λειτουργούν με ρυθμούς δειγματοληψίας έως 1 MSPS ανά κανάλι, ενώ οι ADC1 και ADC2 λειτουργούν σε συγχρονισμένο dual mode. Η μεταφορά των δεδομένων από τους ADC προς τη μνήμη πραγματοποιείται μέσω DMA σε circular double-buffer mode, επιτρέποντας συνεχή συλλογή δεδομένων με ελάχιστη επιβάρυνση του επεξεργαστή. Για την επικοινωνία με τον ηλεκτρονικό υπολογιστή αναπτύχθηκε συσκευή USB ειδικής κλάσης βασισμένη στο πρότυπο USB Full Speed και σε μαζικά τελικά σημεία. Η συγκεκριμένη προσέγγιση επιτρέπει αξιόπιστη μεταφορά δυαδικών δεδομένων και καλύτερη αξιοποίηση του διαθέσιμου εύρους ζώνης USB σε σχέση με μία συμβατική υλοποίηση εικονικής σειριακής θύρας. Παράλληλα αναπτύχθηκε εφαρμογή σε Python για τη λήψη των δεδομένων μέσω WinUSB/libusb, την αποκωδικοποίηση των πακέτων και την απεικόνιση των σημάτων σε πραγματικό χρόνο. Η εφαρμογή αποτελεί τη βάση για περαιτέρω επεξεργασία και ανάλυση των μετρήσεων μέσω εργαλείων επιστημονικού υπολογισμού.

Το τελικό αποτέλεσμα είναι ένα πλήρως λειτουργικό πρωτότυπο σύστημα συλλογής δεδομένων που συνδυάζει ενσωματωμένο υλικό υψηλής απόδοσης με λογισμικό υπολογιστή για τη μεταφορά και οπτικοποίηση των μετρήσεων.

«High-Precision Measurement System with at Least Three Analog Channels and Development of a Computer Application for Measurement Processing»

Andreas Gkikas

Abstract

The purpose of this diploma thesis is the design and implementation of a high-speed data acquisition system featuring three independent analog measurement channels, along with the development of a computer application for data reception and visualization.

The system is based on the STM32H743ZI microcontroller, featuring an ARM Cortex-M7 core, and utilizes its three integrated Analog-to-Digital Converters (ADCs). The ADCs operate at sampling rates of up to 1 MSPS per channel, while ADC1 and ADC2 are configured in synchronized dual mode. Data transfers from the ADC peripherals to memory are performed using DMA in circular double-buffer mode, enabling continuous acquisition with minimal processor intervention. Communication with the host computer is achieved through a custom USB Device Class implementation based on USB Full Speed and bulk endpoints. This approach provides reliable binary data transfer and improved bandwidth utilization compared to a conventional virtual serial port implementation. A Python-based host application was developed to receive measurement data through the WinUSB/libusb software stack, decode incoming packets, and display the acquired signals in real time. The application also provides a foundation for further data analysis and processing using scientific computing tools and Jupyter Notebook environments.

The final result is a fully functional prototype data acquisition system that combines high-performance embedded hardware with a flexible host-side software solution for data transfer, visualization, and future signal analysis applications.

Ευχαριστίες

Ευχαριστώ θερμά τον κ. Άγγελο Γιακουμή, αναπληρωτή καθηγητή του Τμήματος Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙΠΑΕ, για την αμέριστη βοήθεια σε όλη τη διάρκεια της εκπόνησης της εργασίας και την Άρτεμις Κακαλιού για την ηθική υποστήριξη.

Περιεχόμενα

Κατάλογος Σχημάτων

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία
ADC	Analog to Digital Converter / Μετατροπέας Αναλογικού σε Ψηφιακό
API	Application Programming Interface
ARM	Advanced RISC Machines
AXI	Advanced eXtensible Interface
BDMA	Basic Direct Memory Access
CDC	Communications Device Class
CPU	Central Processing Unit / Κεντρική Μονάδα Επεξεργασίας
CRC	Cyclic Redundancy Check
DAC	Digital to Analog Converter / Μετατροπέας Ψηφιακού σε Αναλογικό
DAS	Data Acquisition System / Σύστημα Συλλογής Δεδομένων
D-Cache	Data Cache (Δες I-Cache)
DMA	Direct Memory Access / Άμεση Πρόσβαση Μνήμης
DNL	Differential Non-Linearity
DTCM	Data Tightly Coupled Memory
ENOB	Effective Number Of Bits
FIFO	First In First Out
FFT	Fast Fourier Transform / Γρήγορος Μετασχηματισμός Fourier
FIR	Finite Impulse Response
FS	Full Speed (USB - Δες HS)
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HID	Human Interface Device
HS	High-Speed (USB - FS)
I-Cache	Instruction Cache (Δες D-Cache)
INL	Integral Non-Linearity
IRQ	Interrupt Request
ISR	Interrupt Service Routine

ITCM	Instruction Tightly Coupled Memory
KB	Kilobyte
MB	Megabyte
MCU	Microcontroller Unit / Μικροελεγκτής
MHz	Megahertz
MSPS	Mega Samples Per Second
M7	Cortex-M7
NVIC	Nested Vector Interrupt Controller
PCB	Printed Circuit Board / Πλακέτα Τυπωμένου Κυκλώματος
PC	Personal Computer / Προσωπικός Υπολογιστής
PID	Product Identifier (Δες VID)
PLL	Phase Locked Loop
RAM	Random Access Memory / Μνήμη Τυχαίας Προσπέλασης
RMS	Root Mean Square
SAR ADC	Successive Approximation Register ADC
SRAM	Static Random Access Memory
SNR	Signal to Noise Ratio / Λόγος Σήματος προς Θόρυβο
SPI	Serial Peripheral Interface
ST	STMicroelectronics
SWD	Serial Wire Debug
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VID	Vendor Identifier (Δες PID)
VCP	Virtual COM Port
WinUSB	Windows USB Generic Driver
DMA HT	DMA Half Transfer
DMA TC	DMA Transfer Complete
DSP	Digital Signal Processing / Ψηφιακή Επεξεργασία Σήματος

Κεφάλαιο 1ο: Εισαγωγή

1.1 Γενικά

Η ραγδαία εξέλιξη των ενσωματωμένων ηλεκτρονικών συστημάτων καθώς και των μικροελεγκτών υψηλών επιδόσεων τα τελευταία χρόνια έχει οδηγήσει στην ανάπτυξη συστημάτων συλλογής δεδομένων (data acquisition) ολοένα πιο αποδοτικών αλλά και πιο οικονομικών. Το φάσμα εφαρμογών όπου τα συστήματα αυτά χρησιμοποιούνται διευρύνεται διαρκώς σε πεδία τόσο της βιομηχανικής παραγωγής, όσο και της επιστημονικής έρευνας. Σημαντική πρόοδος έχει σημειωθεί σε τομείς όπως τα συστήματα αυτοματισμού, οι τηλεπικοινωνίες, οι ιατρικές διατάξεις και οι εφαρμογές ψηφιακής επεξεργασίας σημάτων.

Βασικός στόχος ενός συστήματος συλλογής δεδομένων είναι αναλογικά φυσικά μεγέθη να μετατρέπονται σε ψηφιακή μορφή, ώστε ως ψηφιακά πλέον δεδομένα να μπορούν να αποθηκευτούν, να μεταδοθούν και να υποστούν επεξεργασία μέσω ηλεκτρονικού υπολογιστή.

Η αύξηση αφενός της ταχύτητας δειγματοληψίας και αφετέρου των καναλιών που πρέπει να παρακολουθυύνται ταυτόχρονα είναι ανάγκες που αυξάνονται καθημερινά. Αυτό έχει αυξήσει σημαντικά τις απαιτήσεις των σύγχρονων συστημάτων συλλογής. Επιπλέον, σε πολλές εφαρμογές, η απλή λήψη μετρήσεων δεν επαρκεί πλέον· απαιτείται συνεχής και αξιόπιστη μεταφορά ενός ολοένα μεγαλύτερου όγκου δεδομένων σε πραγματικό χρόνο. Η απαίτηση αυτή δημιουργεί σημαντικές τεχνικές δυσκολίες, καθώς το σύστημα πρέπει να συνδυάζει υψηλή υπολογιστική απόδοση, αποδοτική διαχείριση μνήμης και δυνατότητα γρήγορης επικοινωνίας με εξωτερικά συστήματα.

Οι σύγχρονοι μικροελεγκτές υψηλών επιδόσεων, όπως οι μικροελεγκτές αρχιτεκτονικής ARM Cortex-M7, ενσωματώνουν προηγμένα περιφερειακά. Αυτά επιτρέπουν την ανάπτυξη των σχετικών εφαρμογών χωρίς τη χρήση εξειδικευμένου εξωτερικού hardware. Μεταξύ των δυνατοτήτων τους περιλαμβάνονται: μετατροπείς αναλογικού σε ψηφιακό σήμα υψηλής ανάλυσης, μηχανισμοί άμεσης πρόσβασης στη μνήμη (Direct Memory Access – DMA), πολλαπλοί δίαυλοι μνήμης και ενσωματωμένα συστήματα επικοινωνίας USB υψηλής ταχύτητας (USB High Speed). Η αξιοποίηση των δυνατοτήτων αυτών επιτρέπει την υλοποίηση πολύπλοκων ενσωματωμένων (embedded) εφαρμογών χωρίς υψηλό κόστος και διατηρώντας μικρό το μέγεθος συσκευής.

Παράλληλα, η χρήση γλωσσών υψηλού επιπέδου όπως η Python στον υπολογιστή επιτρέπει την ταχεία ανάπτυξη λογισμικού επεξεργασίας δεδομένων, καθώς παρέχει έτοιμες βιβλιοθήκες για αριθμητικούς υπολογισμούς, απεικόνιση σημάτων και ψηφιακή επεξεργασία. Έτσι, δημιουργείται ένα πλήρες σύστημα συλλογής δεδομένων, το οποίο συνδυάζει firmware πραγματικού χρόνου στον μικροελεγκτή και εφαρμογή ανάλυσης δεδομένων στον υπολογιστή.

1.2 Αντικείμενο της εργασίας

Η παρούσα διπλωματική εργασία έχει ως αντικείμενο τη σχεδίαση και υλοποίηση ενός συστήματος συλλογής αναλογικών δεδομένων υψηλής ταχύτητας. Το σύστημα αυτό βασίζεται στον μικροελεγκτή STM32H743ZI της εταιρείας STMicroelectronics. Το σύστημα σχεδιάστηκε με στόχο την ταυτόχρονη

δειγματοληψία τριών ανεξάρτητων αναλογικών καναλιών, με ανάλυση έως 16 bits και ρυθμούς δειγματοληψίας που προσεγγίζουν το 1 MSPS ανά κανάλι.
Για την υλοποίηση της συλλογής δεδομένων αξιοποιούνται και οι τρεις εσωτερικοί μετατροπείς ADC του μικροελεγκτή. Οι ADC1 και ADC2 λειτουργούν σε λειτουργία συγχρονισμού (dual mode), επιτρέποντας ταυτόχρονη δειγματοληψία δύο καναλιών, ενώ ο ADC3 συγχρονίζεται από το ίδιο σκανδαλισμό ενεργοποίησης (trigger event) με τον ADC1 (και κατά συνέπεια με τον ADC2). Η μεταφορά των δεδομένων από τους ADC προς τη μνήμη πραγματοποιείται μέσω DMA, έτσι ώστε η συλλογή των δειγμάτων να γίνεται χωρίς συνεχή παρέμβαση του επεξεργαστή μειώνοντας σημαντικά το υπολογιστικό φορτίο της κεντρικής μονάδας επεξεργασίας (Central Processing Unit – CPU) και επιτυγχάνοντας σταθερή λειτουργία ακόμη και σε υψηλούς ρυθμούς συλλογής δεδομένων. Ένα ακόμα πλεονέκτημα των DMA είναι η αποφυγή διαφθοράς των δεδομένων (data corruption) κατά τη συνεχή δειγματοληψία μέσω της τεχνικής του διπλού buffer (double buffer).
Ιδιαίτερη σημασία δόθηκε στη διαχείριση της ροής δεδομένων και στη βελτιστοποίηση της μεταφοράς μέσω USB. Η μεγάλη ποσότητα παραγόμενων δεδομένων δημιουργεί περιορισμούς ως προς το διαθέσιμο εύρος ζώνης του USB Full Speed interface, γεγονός που επηρέασε σημαντικά τη συνολική αρχιτεκτονική του συστήματος. Για να αντιμετωπιστεί αυτό το πρόβλημα, εξετάστηκαν δύο διαφορετικοί τρόποι λειτουργίας. Στην πρώτη προσέγγιση, τα δεδομένα αποθηκεύονται προσωρινά στη μνήμη RAM του μικροελεγκτή και αποστέλλονται αργότερα στον υπολογιστή. Στη δεύτερη προσέγγιση πραγματοποιείται συνεχές ροή δεδομένων (streaming) μέσω USB για χαμηλότερους ρυθμούς δειγματοληψίας, όπου το διαθέσιμο εύρος ζώνης επαρκεί για μεταφορά σε πραγματικό χρόνο.
Η επικοινωνία με τον υπολογιστή υλοποιήθηκε αρχικά για σκοπούς δοκιμής και ανάπτυξης μέσω USB χρησιμοποιήθηκε CDC (Virtual COM Port), ενώ στη συνέχεια υλοποιήθηκε ειδική κλάση (custom class) USB με τερματικά μαζικής μεταφοράς (bulk endpoints) ώστε να επιτευχθεί υψηλότερη δυνατότητα διεκπεραίωσης των εργασιών με αποτελεσματικότερη διαχείριση των δεδομένων. Στον υπολογιστή αναπτύχθηκε εφαρμογή σε Python για λήψη, αποθήκευση, απεικόνιση και επεξεργασία των μετρήσεων.

1.3 Στόχοι της εργασίας

Βασικός στόχος της εργασίας είναι η ανάπτυξη ενός πλήρους και λειτουργικού συστήματος συλλογής δεδομένων υψηλής ταχύτητας, το οποίο να συνδυάζει αποτελεσματικά το firmware και το λογισμικό επεξεργασίας στον υπολογιστή. Η εργασία δεν αφορά μόνο στην απλή ανάγνωση αναλογικών σημάτων, αλλά επεκτείνεται στη μελέτη και τους τρόπους αντιμετώπισης πραγματικών τεχνικών προβλημάτων που εμφανίζονται γενικότερα σε συστήματα συνεχούς δειγματοληψίας μεγάλου όγκου δεδομένων.

Επιμέρους σκοποί της εργασίας είναι:

- η αξιοποίηση των προηγμένων δυνατοτήτων του μικροελεγκτή STM32H743ZI, όπως οι πολλαπλοί ADC, οι μηχανισμοί DMA και η αρχιτεκτονική υψηλών επιδόσεων ARM Cortex-M7·
- η υλοποίηση firmware πραγματικού χρόνου το οποίο να μπορεί να διαχειρίζεται συνεχείς ροές δεδομένων χωρίς απώλειες δειγμάτων και χωρίς σημαντική επιβάρυνση του επεξεργαστή·
- η διερεύνηση των περιορισμών της διασύνδεσης Full Speed USB σε εφαρμογές συλλογής δεδομένων υψηλής διεκπεραιωτικότητας (high throughput). Η ανάλυση αυτή περιλαμβάνει τόσο θεωρητικούς υπολογισμούς εύρους ζώνης όσο και πειραματικές μετρήσεις της πραγματικής απόδοσης του συστήματος·
- η διερεύνηση διαφορετικών αρχιτεκτονικών μεταφοράς δεδομένων και διαφορετικών στρατηγικών buffering·
- η ανάπτυξη λογισμικού στον υπολογιστή σε γλώσσα Python που να επιτρέπει την εύκολη λήψη και επεξεργασία των δεδομένων. Η εφαρμογή αυτή λειτουργεί ως το

τελικό επίπεδο του acquisition pipeline και επιτρέπει τόσο την οπτικοποίηση των σημάτων όσο και την περαιτέρω ψηφιακή επεξεργασία τους.

1.4 Τεχνικές προκλήσεις

Η ανάπτυξη ενός συστήματος συλλογής δεδομένων υψηλής ταχύτητας συνοδεύεται από σημαντικές τεχνικές προκλήσεις. Η πρώτη και σημαντικότερη αφορά την *αξιόπιστη* συλλογή *μεγάλου* όγκου δεδομένων *χωρίς απώλειες* δειγμάτων. Καθώς οι μετατροπείς ADC λειτουργούν με υψηλούς ρυθμούς δειγματοληψίας, ο επεξεργαστής δεν είναι δυνατόν να εξυπηρετεί κάθε μετατροπή ξεχωριστά με διακοπές (interrupts). Για τον λόγο αυτό χρησιμοποιείται DMA σε λειτουργία κυκλικού διπλού-buffer (circular double-buffer mode), μια τεχνική αποθήκευσης δεδομένων όπου χρησιμοποιούνται εναλλάξ δύο μνήμες, επιτρέποντας τη συνεχή μεταφορά δεδομένων στη μνήμη χωρίς παρέμβαση του CPU.

Η δεύτερη πρόκληση σχετίζεται με την αρχιτεκτονική μνήμης του μικροελεγκτή και τη λειτουργία της προσωρινής μνήμης (cache) στον πυρήνα ARM Cortex-M7. Η χρήση DMA σε συνδυασμό με την data cache μπορεί να δημιουργήσει προβλήματα συνοχής δεδομένων (cache coherency), τα οποία πρέπει να αντιμετωπιστούν μέσω κατάλληλης διαχείρισης της cache ή μέσω χρήσης συγκεκριμένων περιοχών της μνήμης.

Ιδιαίτερο ενδιαφέρον παρουσιάζει επίσης το πρόβλημα του εύρους ζώνης της USB επικοινωνίας. Για τρία κανάλια δειγματοληψίας, ανάλυση 16 bit και ρυθμό 1 MSPS ανά κανάλι, ο συνολικός ρυθμός δεδομένων με βάση τον τύπο:

$$R = N_{ch} \cdot f_s \cdot N_{bits}$$

υπολογίζεται σε :

$$R = 3 \times 10^6 \times 16 = 48\,Mbps$$

Ο ρυθμός αυτός υπερβαίνει σημαντικά την πρακτικά διαθέσιμη δυνατότητα διεκπεραίωσης του Full Speed USB, γεγονός που καθιστά αδύνατη τη συνεχή μεταφορά δεδομένων σε πραγματικό χρόνο στη μέγιστη ταχύτητα δειγματοληψίας. Η παρατήρηση αυτή επηρέασε καθοριστικά τη σχεδίαση του συστήματος και οδήγησε στην ανάπτυξη διαφορετικών τρόπων λειτουργίας και στρατηγικών buffering.

1.5 Δομή της εργασίας

Η εργασία οργανώνεται σε οκτώ κεφάλαια. Στο δεύτερο κεφάλαιο παρουσιάζεται το θεωρητικό υπόβαθρο που σχετίζεται με τους μετατροπείς ADC, τους μηχανισμούς DMA, την αρχιτεκτονική του ARM Cortex-M7 και την τεχνολογία USB. Στο τρίτο κεφάλαιο αναλύεται η συνολική σχεδίαση του συστήματος και η αρχιτεκτονική του hardware. Στο τέταρτο κεφάλαιο παρουσιάζεται η ανάπτυξη του firmware και η λειτουργία των μετατροπέων ADC και DMA. Στο πέμπτο κεφάλαιο περιγράφεται η υλοποίηση της επικοινωνίας μέσω USB και η μετάβαση από CDC σε custom bulk interface. Στο έκτο κεφάλαιο αναλύεται η εφαρμογή επεξεργασίας δεδομένων σε Python και ο τρόπος επικοινωνίας με τη συσκευή. Στο τελευταίο κεφάλαιο παρατίθενται τα συμπεράσματα και πιθανές μελλοντικές επεκτάσεις της εργασίας.

Κεφάλαιο 2ο: Θεωρητικό Υπόβαθρο

2.1 Αρχιτεκτονική ARM Cortex-M7

Ο πυρήνας ARM Cortex-M7 αποτελεί έναν από τους πιο ισχυρούς πυρήνες μικροελεγκτών της οικογένειας ARM Cortex-M και έχει σχεδιαστεί για εφαρμογές υψηλών επιδόσεων πραγματικού χρόνου (real-time embedded systems). Ο πυρήνας αυτός συνδυάζει υψηλή υπολογιστική ισχύ, χαμηλή κατανάλωση ενέργειας και προηγμένους μηχανισμούς διαχείρισης μνήμης και περιφερειακών, που τον καθιστούν ιδιαίτερα κατάλληλο για εφαρμογές επεξεργασίας σημάτων, συστήματα συλλογής δεδομένων υψηλής ταχύτητας και βιομηχανικά ενσωματωμένα συστήματα.

Σε αντίθεση με απλούστερους μικροελεγκτές μικρότερης υπολογιστικής ισχύος, ο Cortex-M7 διαθέτει σημαντικά πιο πολύπλοκη εσωτερική αρχιτεκτονική. Αυτή η αρχιτεκτονική επιτρέπει την εκτέλεση πολλαπλών λειτουργιών παράλληλα και την επίτευξη πολύ υψηλότερων συχνοτήτων λειτουργίας, χωρίς να θυσιάζεται η ντετερμινιστική (deterministic) συμπεριφορά που απαιτείται σε εφαρμογές πραγματικού χρόνου.

Ο μικροελεγκτής STM32H743ZI που χρησιμοποιείται στην παρούσα εργασία βασίζεται στον πυρήνα ARM Cortex-M7 και λειτουργεί σε συχνότητες έως και 480 MHz. Η υπολογιστική αυτή ισχύς είναι ιδιαίτερα σημαντική για την υλοποίηση συστήματος συλλογής δεδομένων υψηλής ταχύτητας, καθώς επιτρέπει την ταυτόχρονη λειτουργία πολλαπλών ADC, DMA μεταφορών, USB επικοινωνίας και επεξεργασίας δεδομένων, χωρίς σημαντική επιβάρυνση του επεξεργαστή.

2.1.1 Αρχιτεκτονική Harvard

Ο Cortex-M7 βασίζεται σε τροποποιημένη αρχιτεκτονική Harvard.[1] Στην αρχιτεκτονική αυτή οι δίαυλοι εντολών και δεδομένων είναι ανεξάρτητοι, κάνοντας δυνατή για τον επεξεργαστή την *ταυτόχρονη* προσπέλαση τόσο σε εντολές προγράμματος όσο και σε δεδομένα μνήμης.

Η δυνατότητα αυτή αυξάνει σημαντικά την απόδοση του συστήματος σε σχέση με την κλασική αρχιτεκτονική Von Neumann, όπου εντολές και δεδομένα μοιράζονται τον *ίδιο* δίαυλο.

Στην πράξη, αυτό σημαίνει ότι ο επεξεργαστής μπορεί να εκτελεί εντολές ενώ παράλληλα πραγματοποιούνται προσπελάσεις δεδομένων από DMA ή περιφερειακά, μειώνοντας τα σημεία συμφόρησης της μνήμης (τα λεγόμενα bottlenecks) και αυξάνοντας τη συνολική δυνατότητα διεκπεραίωσης εργασιών του συστήματος.

2.1.2 Διοχέτευση εντολών επεξεργασίας

Ο Cortex-M7 χρησιμοποιεί πολυσταδιακή διοχέτευση εντολών (pipeline) επεξεργασίας, η οποία επιτρέπει την ταυτόχρονη εκτέλεση διαφορετικών σταδίων εντολών. Αντί κάθε εντολή να ολοκληρώνεται πλήρως πριν ξεκινήσει η επόμενη, πολλές εντολές βρίσκονται ταυτόχρονα σε διαφορετικά στάδια εκτέλεσης.

Η τεχνική αυτή αυξάνει σημαντικά την απόδοση του επεξεργαστή διότι επιτρέπει την εκτέλεση μεγάλου πλήθους εντολών ανά δευτερόλεπτο. Ο Cortex-M7 διαθέτει superscalar pipeline, το οποίο μπορεί υπό

[1] Η αρχιτεκτονική Harvard είναι ιδιαίτερα σημαντική σε εφαρμογές συλλογής δεδομένων υψηλής ταχύτητας, καθώς μεγάλος όγκος δεδομένων μεταφέρεται συνεχώς μεταξύ ADC, DMA και RAM ενώ ταυτόχρονα εκτελείται firmware πραγματικού χρόνου.

συγκεκριμένες συνθήκες να εκτελεί περισσότερες από μία εντολές ανά κύκλο ρολογιού. Η χρήση pipeline όμως εισάγει και πρόσθετη πολυπλοκότητα, ιδιαίτερα σε περιπτώσεις εντολών διακλάδωσης ή κινδύνου μνήμης (memory hazards). Για την αντιμετώπιση αυτών των προβλημάτων χρησιμοποιούνται μηχανισμοί πρόβλεψης διακλάδωσης και προανάκτησης εντολών.

Σε ενσωματωμένες εφαρμογές πραγματικού χρόνου, όπως αυτή της παρούσας εργασίας, η υψηλή απόδοση του pipeline είναι ιδιαίτερα χρήσιμη επειδή:

- μειώνει τον χρόνο εξυπηρέτησης διακοπών (interrupts)
- αυξάνει την ταχύτητα επεξεργασίας δεδομένων·
- επιτρέπει τη λειτουργία πολλών υποσυστημάτων ταυτόχρονα.

2.1.3 Cache μνήμης

Ένα από τα σημαντικότερα χαρακτηριστικά του Cortex-M7 είναι η ύπαρξη cache μνήμης. Ο πυρήνας διαθέτει:

- προσωρινή μνήμη εντολών (instruction cache, I-Cache),
- και προσωρινή μνήμη δεδομένων (data cache, D-Cache).

Η cache αποτελεί μικρή αλλά εξαιρετικά γρήγορη μνήμη η οποία χρησιμοποιείται για την προσωρινή αποθήκευση συχνά χρησιμοποιούμενων δεδομένων και εντολών. Με αυτό τον τρόπο μειώνεται σημαντικά ο αριθμός προσπελάσεων προς τη βασική μνήμη SRAM ή Flash, αυξάνοντας την απόδοση του συστήματος.

Η μεν προσωρινή μνήμη εντολών αποθηκεύει εντολές προγράμματος που χρησιμοποιούνται συχνά (μειώνοντας έτσι τον χρόνο ανάγνωσης από τη μνήμη flash), ενώ η προσωρινή μνήμη δεδομένων αποθηκεύει δεδομένα που χρησιμοποιούνται κατ' επανάληψη από τον επεξεργαστή.

Παρότι η χρήση της cache αυξάνει σημαντικά την απόδοση, δημιουργεί προβλήματα συνοχής δεδομένων (cache coherency) όταν χρησιμοποιείται DMA. Το DMA μπορεί να μεταβάλλει περιοχές της μνήμης χωρίς ο επεξεργαστής να ενημερώνεται άμεσα για τις αλλαγές αυτές. Ως αποτέλεσμα, η cache μπορεί να περιέχει παλαιά δεδομένα, τα οποία δεν αντιστοιχούν στο πραγματικό περιεχόμενο της RAM.

Το πρόβλημα αυτό είναι ιδιαίτερα σημαντικό στην παρούσα εργασία, καθώς οι ADC μεταφέρουν συνεχώς δεδομένα στη μνήμη μέσω DMA. Για την αποφυγή ασυνεπειών στα δεδομένα απαιτείται:

- invalidate της cache μετά από DMA reads,
- clean operations πριν από DMA writes,
- ή χρήση non-cacheable περιοχών μνήμης.

Η σωστή διαχείριση της cache αποτελεί κρίσιμο παράγοντα για την αξιοπιστία συστημάτων υψηλής διεκπεραιωτικότητας.

2.1.4 Διαχείριση μνήμης και δίαυλοι

Ο Cortex-M7 διαθέτει πολύ πιο σύνθετη αρχιτεκτονική μνήμης σε σχέση με παλαιότερους μικροελεγκτές. Η μνήμη χωρίζεται σε διαφορετικές περιοχές με διαφορετικά χαρακτηριστικά ταχύτητας και πρόσβασης.

Στον STM32H743ZI υπάρχουν:

- AXI SRAM,
- DTCM RAM,
- ITCM RAM,
- SRAM περιοχές διαφορετικών domains,
- καθώς και πολλαπλοί εσωτερικοί δίαυλοι.

Η ύπαρξη πολλών memory buses επιτρέπει την παράλληλη λειτουργία CPU, DMA και περιφερειακών. Για παράδειγμα, ενώ ο επεξεργαστής εκτελεί κώδικα από μία περιοχή μνήμης, οι DMA controllers μπορούν ταυτόχρονα να μεταφέρουν δεδομένα ADC προς διαφορετική SRAM περιοχή.

Η επιλογή της σωστής περιοχής μνήμης επηρεάζει σημαντικά:

- το throughput,
- τη latency,
- και τη σταθερότητα του συστήματος.

Σε εφαρμογές acquisition υψηλής ταχύτητας είναι σημαντικό τα DMA buffers να τοποθετούνται σε περιοχές μνήμης προσβάσιμες από τους DMA controllers και κατά προτίμηση εκτός cache ή με σωστή cache synchronization.

2.1.5 Σύστημα interrupts και NVIC

Ο Cortex-M7 διαθέτει προηγμένο σύστημα διακοπών μέσω του Nested Vector Interrupt Controller (NVIC). Το σύστημα αυτό επιτρέπει:

- nested interrupts,
- programmable priorities,
- fast interrupt response,
- και deterministic handling πραγματικού χρόνου.

Η δυνατότητα nested interrupts σημαίνει ότι interrupts υψηλότερης προτεραιότητας μπορούν να διακόψουν την εξυπηρέτηση χαμηλότερης προτεραιότητας interrupt. Αυτό είναι ιδιαίτερα σημαντικό σε συστήματα acquisition όπου συγκεκριμένα γεγονότα, όπως DMA half-transfer interrupts ή USB events, πρέπει να εξυπηρετούνται άμεσα.

Ο NVIC υποστηρίζει μεγάλο αριθμό interrupt sources και παρέχει μηχανισμούς γρήγορης αποθήκευσης context κατά την είσοδο σε interrupt handler. Με τον τρόπο αυτό μειώνεται η interrupt latency και βελτιώνεται η deterministic συμπεριφορά του συστήματος.

Στην παρούσα εργασία οι μηχανισμοί interrupts χρησιμοποιούνται κυρίως για:

- ειδοποίηση ολοκλήρωσης DMA μεταφορών,
- συγχρονισμό acquisition buffers,
- και διαχείριση USB events.

2.1.6 DMA και παράλληλη λειτουργία περιφερειακών

Η χρήση DMA αποτελεί βασικό στοιχείο της αρχιτεκτονικής Cortex-M7. Οι DMA controllers επιτρέπουν τη μεταφορά δεδομένων μεταξύ περιφερειακών και μνήμης χωρίς συνεχή συμμετοχή του επεξεργαστή.

Στο σύστημα της παρούσας εργασίας χρησιμοποιούνται:

- DMA1,
- DMA2,
- και BDMA,
  για τη μεταφορά δεδομένων από τους ADC προς τη RAM.

Η χρήση DMA είναι απαραίτητη, καθώς σε ρυθμούς δειγματοληψίας της τάξης του 1 MSPS ο επεξεργαστής δεν θα μπορούσε να εξυπηρετεί κάθε ADC conversion ξεχωριστά μέσω software polling ή interrupts.

Η λειτουργία circular double-buffer mode επιτρέπει τη συνεχή συλλογή δεδομένων χωρίς διακοπή της δειγματοληψίας. Ενώ το DMA γεμίζει το ένα μισό του buffer, ο επεξεργαστής μπορεί να επεξεργάζεται ή να μεταδίδει το άλλο μισό. Με αυτόν τον τρόπο επιτυγχάνεται παράλληλη λειτουργία acquisition και communication subsystems.

Η δυνατότητα αυτή είναι ιδιαίτερα σημαντική σε embedded συστήματα υψηλού throughput και αποτελεί έναν από τους βασικούς λόγους επιλογής του Cortex-M7 για την παρούσα εργασία.

2.1.7 Συμπεράσματα

Ο πυρήνας ARM Cortex-M7 παρέχει ένα ιδιαίτερα ισχυρό και ευέλικτο περιβάλλον για την ανάπτυξη embedded εφαρμογών υψηλών επιδόσεων. Η ύπαρξη cache, πολλαπλών διαύλων μνήμης, προηγμένου pipeline, DMA controllers και ισχυρού συστήματος interrupts επιτρέπει την υλοποίηση σύνθετων εφαρμογών πραγματικού χρόνου με υψηλές απαιτήσεις throughput.

Οι δυνατότητες αυτές καθιστούν τον STM32H743ZI κατάλληλη πλατφόρμα για την ανάπτυξη του συστήματος acquisition της παρούσας εργασίας, όπου απαιτείται ταυτόχρονη λειτουργία ADC, DMA και USB επικοινωνίας με μεγάλο όγκο δεδομένων και αυστηρούς χρονικούς περιορισμούς.

2.2 Θεωρία Μετατροπέων Αναλογικού σε Ψηφιακό Σήμα (ADC)

Ένα από τα σημαντικότερα υποσυστήματα σε κάθε σύστημα συλλογής (αναλογικών) δεδομένων είναι οι μετατροπείς αναλογικού σε ψηφιακό σήμα (Analog to Digital Converters – ADCs) αποτελούν. Ο ρόλος τους είναι η μετατροπή ενός συνεχούς αναλογικού σήματος σε διακριτές ψηφιακές τιμές, έτσι ώστε συστήματα όπως οι μικροελεγκτές και οι ηλεκτρονικοί υπολογιστές να μπορούν να αποθηκεύσουν, να μεταδώσουν και να επεξεργαστούν το ψηφιακό πλέον σήμα.

Τα περισσότερα φυσικά μεγέθη εμφανίζονται με αναλογική μορφή. Για παράδειγμα, η τάση εξόδου ενός αισθητήρα θερμοκρασίας, το σήμα ενός μικροφώνου, η έξοδος ενός φωτοαισθητήρα ή η τάση ενός κυκλώματος μέτρησης αποτελούν συνεχή αναλογικά σήματα. Δεδομένου όμως ότι οι μικροελεγκτές μπορούν να επεξεργάζονται αποκλειστικά και μόνο ψηφιακά δεδομένα, απαιτείται η χρήση ενός μετατροπέα ADC. Αυτός θα λειτουργήσει ως η «γέφυρα» μεταξύ του αναλογικού και του ψηφιακού κόσμου.

Στην παρούσα εργασία οι μετατροπείς ADC χρησιμοποιούνται για την ταυτόχρονη δειγματοληψία τριών ανεξάρτητων αναλογικών καναλιών υψηλής ταχύτητας μέσω του μικροελεγκτή STM32H743ZI. Πριν την παρουσίαση αυτής της χρήσης των μετατροπέων είναι απαραίτητο να αναφερθούμε στον τρόπο λειτουργίας των ADC γενικά. Η κατανόηση των βασικών αρχών λειτουργίας είναι απαραίτητη τόσο για την καταρχήν σχεδίαση του υποσυστήματος συλλογής δεδομένων όσο και για την αξιολόγηση της ποιότητας των μετρήσεων στη συνέχεια.

2.2.1 Διαδικασία ψηφιοποίησης

Η διαδικασία μετατροπής ενός αναλογικού σήματος σε ψηφιακό (ψηφιοποίηση) αποτελείται από δύο βασικά στάδια:

- τη δειγματοληψία (sampling),
- και την κβάντιση (quantization).

Κατά το στάδιο της δειγματοληψίας, γίνονται μετρήσεις του συνεχούς αναλογικού σήματος σε συγκεκριμένες χρονικές *στιγμές*. Το αποτέλεσμα είναι μια ακολουθία διακριτών χρονικά *τιμών*. Η συχνότητα με την οποία πραγματοποιούνται οι μετρήσεις ονομάζεται *συχνότητα δειγματοληψίας*.

Εάν η συχνότητα δειγματοληψίας είναι αρκετά υψηλή, τότε το αρχικό σήμα μπορεί να ανακατασκευαστεί με μεγάλη ακρίβεια από τα ψηφιακά δείγματα. Συγκεκριμένα, σύμφωνα με το θεώρημα Nyquist-Shannon [1], η συχνότητα δειγματοληψίας πρέπει να είναι τουλάχιστον διπλάσια από τη μέγιστη συχνότητα του σήματος:

$$f_s \geq 2f_{max}$$

όπου:

- f_s είναι η συχνότητα δειγματοληψίας,
- και f_{max} η μέγιστη συχνότητα του αναλογικού σήματος.

Αν η συνθήκη αυτή δεν ικανοποιείται, εμφανίζεται το φαινόμενο της επικάλυψης (aliasing), κατά το οποίο υψηλές συχνότητες εμφανίζονται (λανθασμένα) ως χαμηλότερες στο ψηφιακό σήμα. Η επικάλυψη αποτελεί ένα από τα σημαντικότερα προβλήματα σε συστήματα συλλογής δεδομένων.[2]

Μετά τη δειγματοληψία ακολουθεί το στάδιο της *κβάντισης*. Κατά το στάδιο αυτό κάθε αναλογική τιμή αντιστοιχίζεται στην πλησιέστερη διαθέσιμη ψηφιακή *στάθμη* (τιμή εξόδου), καθώς όμως ο αριθμός των ψηφιακών τιμών είναι πεπερασμένος, η διαδικασία αυτή εισάγει αναπόφευκτα σφάλμα στην απόδοση του αναλογικών τιμών, γνωστό ως σφάλμα *κβάντισης*.

2.2.2 Ανάλυση ADC και κβάντιση

Η ακρίβεια της ανάλυσης που παράγει ένας μετατροπέας ADC καθορίζεται από τον αριθμό των bit που χρησιμοποιούνται για την αναπαράσταση του κάθε δείγματος. Ένας ADC ανάλυσης Nbit μπορεί να παράγει 2^N διαφορετικές ψηφιακές στάθμες. Για παράδειγμα, ένας ADC 16 bit διαθέτει $2^{16} = 65536$ διακριτές τιμές εξόδου.

[2] Για την αποφυγή της επικάλυψης χρησιμοποιούνται συνήθως αναλογικά anti-aliasing φίλτρα πριν τον ADC.

Όσο μεγαλύτερη είναι η ανάλυση, τόσο μικρότερο είναι το βήμα κβάντισης και τόσο μεγαλύτερη η θεωρητική ακρίβεια μέτρησης. Το μέγεθος του μικρότερου διακριτού βήματος ονομάζεται Least Significant Bit (LSB) και υπολογίζεται από τον τύπο:

$$LSB = \frac{V_{ref}}{2^N}$$

όπου:

- V_{ref}είναι η τάση αναφοράς,
- και Nη ανάλυση του ADC.

Για παράδειγμα, για τάση αναφοράς 3.3 V και ανάλυση 16 bit, το θεωρητικό βήμα κβάντισης είναι της τάξης μερικών δεκάδων microvolts. Στην πράξη όμως η πραγματική ακρίβεια είναι ακόμα μικρότερη λόγω θορύβου, offset errors και μη ιδανικής λειτουργίας του ADC.

Επίσης, πρέπει να σημειωθεί ότι η κβάντιση εισάγει τον λεγόμενο θόρυβο κβάντισης (quantization noise), ο οποίος είναι συνήθως ομοιόμορφα κατανεμημένος. Ο θεωρητικός λόγος του σήματος προς τον θόρυβο κβάντισης δίνεται από τον τύπο:

$$SNR = 6.02N + 1.76\ dB$$

Επομένως, κάθε επιπλέον bit ανάλυσης αυξάνει θεωρητικά το SNR, και συγκεκριμένα κατά περίπου 6 dB.

2.2.3 Συχνότητα δειγματοληψίας και throughput

Η συχνότητα δειγματοληψίας καθορίζει πόσα δείγματα ανά δευτερόλεπτο μπορούν να παραχθούν από τον μετατροπέα ADC. Σε εφαρμογές συλλογής δεδομένων υψηλής ταχύτητας η παράμετρος αυτή είναι ιδιαίτερα σημαντική, καθώς επηρεάζει άμεσα:

(1) το *εύρος ζώνης* (bandwidth) του συστήματος·

(2) τον απαιτούμενο ρυθμό μεταφοράς δεδομένων·

(3) τις απαιτήσεις μνήμης.

Στην παρούσα εργασία χρησιμοποιούνται τρία κανάλια δειγματοληψίας με ρυθμούς έως 1 MSPS ανά κανάλι. Ο συνολικός ρυθμός δεδομένων υπολογίζεται από τον τύπο:

$$R = N_{ch} \cdot f_s \cdot N_{bits}$$

Για τρία κανάλια και 16-bit samples προκύπτει ότι ο όγκος των δεδομένων θα είναι:

$$R = 3 \times 10^6 \times 16 = 48\ Mbps$$

Ο όγκος αυτός είναι ιδιαίτερα μεγάλος για ένα ενσωματωμένο σύστημα και δημιουργεί σημαντικές απαιτήσεις τόσο στη μνήμη όσο και στη διεπαφή επικοινωνίας USB.

2.2.4 Χρόνος μετατροπής

Η διαδικασία μετατροπής ενός αναλογικού σήματος σε ψηφιακή μορφή απαιτεί συγκεκριμένο χρονικό διάστημα, γνωστό ως *χρόνος μετατροπής* (conversion time), ο οποίος εξαρτάται από τους εξής παράγοντες:

- την αρχιτεκτονική του ADC,

- τη συχνότητα λειτουργίας,
- τον χρόνο δειγματοληψίας (sample time),
- και την ανάλυση.

Σε ADC υψηλής ταχύτητας είναι σημαντικό ο χρόνος μετατροπής να είναι αρκετά μικρός καθώς μόνο με αυτό τον τρόπο μπορούν να υποστηρίζονται μεγάλοι ρυθμοί δειγματοληψίας.

Στους ADC του STM32H743ZI ο συνολικός χρόνος μετατροπής περιλαμβάνει:

- τον χρόνο δειγματοληψίας του πυκνωτή δειγματοληψίας και συγκράτησης (sample-and-hold),
- και τον χρόνο ψηφιακής μετατροπής.

Η σωστή επιλογή του χρόνου δειγματοληψίας είναι ιδιαίτερα σημαντική. Μικρός χρόνος δειγματοληψίας επιτρέπει υψηλότερο ρυθμό/συχνότητα της δειγματοληψίας, αλλά την ίδια στιγμή είναι δυνατό να μειώσει την ακρίβεια όταν η πηγή του σήματος έχει υψηλή αντίσταση εξόδου.

2.2.5 Κυκλώματα δειγματοληψίας και συγκράτησης

Οι περισσότεροι μετατροπείς ADC χρησιμοποιούν το λεγόμενο κύκλωμα δειγματοληψίας και συγκράτησης (sample-and-hold). Το κύκλωμα αυτό αποθηκεύει προσωρινά την αναλογική τάση σε έναν πυκνωτή πριν ξεκινήσει η μετατροπή ώστε η τάση να παραμένει σταθερή κατά τη διάρκεια καθαυτής της διαδικασίας μετατροπής.

Η λειτουργία sample-and-hold είναι ιδιαίτερα σημαντική σε γρήγορα μεταβαλλόμενα σήματα, καθώς χωρίς αυτήν η τάση εισόδου θα μπορούσε να αλλάξει κατά τη διάρκεια της μετατροπής, οδηγώντας σε λανθασμένο αποτέλεσμα.

Η φόρτιση του πυκνωτή δειγματοληψίας εξαρτάται από:

- την αντίσταση εξόδου της πηγής,
- τη χωρητικότητα του πυκνωτή,
- και τον διαθέσιμο χρόνο δειγματοληψίας.

Για τον λόγο αυτό σε εφαρμογές υψηλής ακρίβειας συχνά χρησιμοποιούνται οι λεγόμενοι *ενισχυτές απομόνωσης* (buffer amplifiers) ή αναλογικά στάδια οδήγησης **χαμηλής αντίστασης**.

2.2.6 Αρχιτεκτονικές ADC

Υπάρχουν πολλές διαφορετικές αρχιτεκτονικές ADC, καθεμία με διαφορετικά χαρακτηριστικά ως προς:

- την ταχύτητα,
- την κατανάλωση,
- και την ακρίβεια.

Οι πιο διαδεδομένες αρχιτεκτονικές είναι οι εξής:

- Flash ADC,
- Sigma-Delta ADC,
- Pipeline ADC,

- και SAR (Successive Approximation Register) ADC.

Οι μετατροπείς ADC των μικροελεγκτών STM32 βασίζονται κυρίως σε αρχιτεκτονική SAR. Η αρχιτεκτονική αυτή προσφέρει εξαιρετική ισορροπία μεταξύ:

- ταχύτητας,
- κατανάλωσης,
- πολυπλοκότητας,
- και ανάλυσης.

Στον μετατροπέα SAR ADC η μετατροπή πραγματοποιείται μέσω διαδοχικών προσεγγίσεων της εισόδου, χρησιμοποιώντας εσωτερικό μετατροπέα DAC και συγκριτικό μετρητή (comparator). Η διαδικασία ξεκινά από το πιο σημαντικό bit και προχωρά προς το λιγότερο σημαντικό.

Η αρχιτεκτονική SAR χρησιμοποιείται ιδιαίτερα στις ενσωματωμένες εφαρμογές συλλογής δεδομένων, καθώς είναι κατάλληλη για να πετύχει κανείς αρκετά υψηλές ταχύτητες δειγματοληψίας με σχετικά χαμηλή κατανάλωση ισχύος.

2.2.7 Σφάλματα των μετατροπέων ADC

Οι πραγματικοί μετατροπείς ADC δεν είναι ωστόσο ιδανικοί και παρουσιάζουν διάφορα σφάλματα που επηρεάζουν την ποιότητα των μετρήσεων.

Ένα βασικό σφάλμα είναι το λεγόμενο *σφάλμα μετατόπισης* (offset error), κατά το οποίο η έξοδος του μετατροπέα παρουσιάζει σταθερή μετατόπιση ακόμη και όταν η είσοδος είναι μηδενική. Επιπλέον υπάρχει *το σφάλμα ενίσχυσης* (gain error), το οποίο επηρεάζει τη συνολική κλίμακα μετατροπής.

Μειονέκτημα αποτελεί επίσης η μη γραμμικότητα του μετατροπέα ADC, η οποία διακρίνεται σε: (α) *διαφορική μη γραμμικότητα* (Differential Non-Linearity, DNL) η οποία αφορά αποκλίσεις στο μέγεθος των διαδοχικών βημάτων κβάντισης, και (β) *ολοκληρωτική μη γραμμικότητα* (Integral Non-Linearity, INL), που εκφράζει τη συνολική απόκλιση της χαρακτηριστικής μεταφοράς από την ιδανική ευθεία.

Τέλος, σε πραγματικές συνθήκες λειτουργίας εμφανίζεται επίσης ηλεκτρικός θόρυβος από:

- το εσωτερικό κύκλωμα του ADC,
- το τροφοδοτικό,
- το PCB,
- και το εξωτερικό περιβάλλον.

Ο θόρυβος αυτός μειώνει την πραγματική ανάλυση του ADC, που μας οδηγεί στην έννοια του πραγματικού αριθμού χρήσιμων bit (Effective Number of Bits, ENOB).

2.2.8 Ταυτόχρονη δειγματοληψία

Σε πολυκαναλικά συστήματα συλλογής δεδομένων είναι συχνά απαραίτητη η σχεδόν ταυτόχρονη μέτρηση πολλών σημάτων. Η απαίτηση αυτή εμφανίζεται ιδιαίτερα σε εφαρμογές:

- ψηφιακής επεξεργασίας σημάτων,
- συστημάτων ελέγχου,
- και ανάλυσης πολυφασικών σημάτων.

Στην παρούσα εργασία οι μετατροπείς ADC1 και ADC2 λειτουργούν με διπλό συγχρονισμό (dual simultaneous mode), με συγχρονισμένη δηλαδή δειγματοληψία δύο καναλιών μέσω σκανδάλης hardware. Χάρη σε αυτή τη δυνατότητα μειώνεται σημαντικά η χρονική στρέβλωση (skew) μεταξύ των μετρήσεων και βελτιώνεται γενικότερα η χρονική ακρίβεια του συστήματος. Η χρήση του hardware triggering επιπλέον εξασφαλίζει ακριβή χρονισμό ανεξάρτητα από τυχόν διακοπές ή καθυστερήσεις στον χρόνο απόκρισης (latency) του software.

2.2.9 Συμπεράσματα

Λαμβάνοντας υπόψη όλα τα παραπάνω, οι μετατροπείς ADC αποτελούν το βασικότερο υποσύστημα κάθε συστήματος συλλογής (αναλογικών) δεδομένων. Επηρεάζουν άμεσα την ποιότητα, την ακρίβεια και την ταχύτητα των μετρήσεων. Παράμετροι όπως η ανάλυση, η συχνότητα δειγματοληψίας, ο χρόνος μετατροπής και τα σφάλματα κβάντισης καθορίζουν τις δυνατότητες αλλά και τους περιορισμούς κάθε συστήματος συλλογής δεδομένων.

Η σωστή κατανόηση της λειτουργίας των μετατροπέων ADC είναι απαραίτητη για την αποδοτική αξιοποίηση των δυνατοτήτων του STM32H743ZI, ιδιαίτερα σε εφαρμογές υψηλής διεκπεραιωτικότητας εργασιών όπου συνδυάζονται πολλαπλά κανάλια, DMA μεταφορές και επικοινωνία USB σε πραγματικό χρόνο.

2.3 Direct Memory Access (DMA)

Το Direct Memory Access (DMA) αποτελεί έναν από τους σημαντικότερους μηχανισμούς σύγχρονων μικροελεγκτών και ενσωματωμένων συστημάτων υψηλής απόδοσης γιατί η βασική λειτουργία του συνίσταται στην αυτόματη μεταφορά δεδομένων μεταξύ περιφερειακών και μνήμης χωρίς τη συνεχή εμπλοκή του επεξεργαστή, χαρακτηριστικό κρίσιμο σε εφαρμογές πραγματικού χρόνου και σε συστήματα συλλογής δεδομένων υψηλής ταχύτητας, όπου παράγεται μεγάλος όγκος δεδομένων σε μικρό χρονικό διάστημα.

Σε ένα απλό σύστημα χωρίς DMA, κάθε μεταφορά δεδομένων πραγματοποιείται από τον επεξεργαστή. Για παράδειγμα, όταν ένας ADC ολοκληρώνει μία μετατροπή, ο CPU πρέπει: (α) να ανιχνεύσει ότι το conversion ολοκληρώθηκε, (β) να διαβάσει το αποτέλεσμα από το peripheral register, (γ) να το αποθηκεύσει στη RAM. Η διαδικασία αυτή επαναλαμβάνεται για κάθε δείγμα. Σε χαμηλούς ρυθμούς δειγματοληψίας η προσέγγιση αυτή είναι αποδεκτή. Όμως, όταν οι ρυθμοί acquisition αυξάνονται σε εκατοντάδες χιλιάδες ή εκατομμύρια δείγματα ανά δευτερόλεπτο, ο επεξεργαστής αδυνατεί να εξυπηρετήσει αποτελεσματικά τις μεταφορές.

Το DMA δίνει τη λύση στο πρόβλημα αυτό λειτουργώντας ως ανεξάρτητος hardware controller μεταφοράς δεδομένων. Συγκεκριμένα, ο επεξεργαστής ρυθμίζει αρχικά: τη διεύθυνση πηγής, τη διεύθυνση προορισμού, το μέγεθος του buffer, και τον τρόπο λειτουργίας. Μετά την αρχικοποίηση, ο DMA controller πραγματοποιεί αυτόνομα τις μεταφορές χωρίς περαιτέρω παρέμβαση του CPU.

Στην παρούσα εργασία το DMA χρησιμοποιείται για τη μεταφορά των conversion results των ADC προς τη RAM του μικροελεγκτή STM32H743ZI. Η χρήση DMA είναι απαραίτητη λόγω του υψηλού ρυθμού δειγματοληψίας και του μεγάλου όγκου δεδομένων που παράγονται από τα τρία κανάλια συλλογής.

2.3.1 Βασική λειτουργία DMA

Η λειτουργία ενός DMA controller βασίζεται στη δυνατότητα απευθείας πρόσβασης στη μνήμη και στα peripheral registers μέσω των εσωτερικών διαύλων του μικροελεγκτή. Όταν ένα περιφερειακό, όπως ένας ADC, παράγει νέα δεδομένα, δημιουργείται DMA request event. Ο ελεγκτής DMA αποκρίνεται στο συμβάν και πραγματοποιεί αυτόματα αφενός την ανάγνωση από τον καταχωρητή του περιφερειακού, αφετέρου την εγγραφή στη μνήμη. Η διαδικασία αυτή πραγματοποιείται πλήρως σε επίπεδο hardware. Ο CPU απαλλάσσεται από τη συνεχή διαχείριση των μεταφορών και μπορεί να ασχοληθεί με την επεξεργασία δεδομένων, την επικοινωνία USB, ή άλλες λειτουργίες του firmware. Έτσι, η χρήση DMA μειώνει σημαντικά τον φόρτο CPU, τον αριθμό διακοπών, και τη χρονική αβεβαιότητα του συστήματος, πλεονεκτήματα ιδιαίτερα σημαντικά σε συστήματα πραγματικού χρόνου, όπου απαιτείται ντετερμινιστική συμπεριφορά και σταθερός χρονισμός.

2.3.2 DMA σε συστήματα acquisition

Σε συστήματα συλλογής δεδομένων, οι ADC παράγουν συνεχώς νέα δείγματα με σταθερό ρυθμό. Αν κάθε δείγμα εξυπηρετούνταν μέσω interrupt, τότε ο επεξεργαστής θα έπρεπε να εξυπηρετεί τεράστιο αριθμό interrupts ανά δευτερόλεπτο.

Για παράδειγμα, σε ρυθμό δειγματοληψίας 1 MSPS (1 Mega Sample Per Second), παράγονται: 10^6 samples/s για κάθε κανάλι. Σε τρία κανάλια ο συνολικός ρυθμός γίνεται: 3×10^6 samples/s.

Η εξυπηρέτηση εκατομμυρίων interrupts ανά δευτερόλεπτο είναι πρακτικά αδύνατη για έναν embedded επεξεργαστή, ακόμη και για έναν ισχυρό πυρήνα όπως ο ARM Cortex-M7. Η χρήση DMA επιτρέπει τη μεταφορά δεδομένων σε μεγάλα μπλοκ αντί για μεμονωμένα δείγματα. Με τον τρόπο αυτό μειώνεται δραστικά το interrupt overhead και αυξάνεται σημαντικά η συνολική αποδοτικότητα του συστήματος.

2.3.3 Circular mode

Ένα από τα σημαντικότερα χαρακτηριστικά του DMA σε εφαρμογές acquisition είναι η κυκλική λειτουργία (circular mode). Σε μια απλή μεταφορά DMA, όταν ολοκληρωθεί η πλήρωση του buffer, ο DMA σταματά να λειτουργεί. Η λειτουργία αυτή είναι κατάλληλη για one-shot μεταφορές αλλά όχι για συνεχή συλλογή. Στο circular mode ο DMA δεν σταματά όταν φτάσει στο τέλος του buffer. Αντίθετα, επιστρέφει αυτόματα στην αρχή του buffer και συνεχίζει να γράφει νέα δεδομένα, συμπεριφορά που επιτρέπει τη δημιουργία ενός συνεχούς pipeline συλλογής χωρίς ανάγκη συνεχούς επανεκκίνησης του DMA από το firmware. Η τεχνική αυτή είναι ιδιαίτερα σημαντική σε εφαρμογές συνεχούς παρακολούθησης, streaming, και επεξεργασίας σε πραγματικό χρόνο (real-time processing).

Ο επεξεργαστής δεν χρειάζεται να ασχολείται με την επαναρύθμιση του DMA μετά από κάθε μεταφορά, γεγονός που όχι μόνο μειώνει το φόρτο εργασίας του software, αλλά και αυξάνει τη σταθερότητα και τη ντετερμινιστική συμπεριφορά του συστήματος. Στην πράξη, το circular mode μετατρέπει τον buffer σε κυκλική περιοχή μνήμης μέσα στην οποία τα νέα δεδομένα αντικαθιστούν σταδιακά τα παλαιότερα. Η χρήση circular buffers είναι ιδιαίτερα αποδοτική σε ενσωματωμένα συστήματα περιορισμένης μνήμης, καθώς επιτρέπει τη συνεχή επαναχρησιμοποίηση του ίδιου memory region.

2.3.4 Double buffering

Παρότι η κυκλική λειτουργία επιτρέπει συνεχή συλλογή, δημιουργεί ένα σημαντικό πρόβλημα: ο DMA μπορεί να γράφει νέα δεδομένα σε μια περιοχή μνήμης την ίδια στιγμή που ο επεξεργαστής προσπαθεί να την επεξεργαστεί. Για την επίλυση του προβλήματος χρησιμοποιείται η τεχνική double buffering, κατά την οποία συμβαίνει το εξής: καταρχήν ο buffer χωρίζεται λογικά σε δύο ανεξάρτητα τμήματα, το πρώτο μισό και το δεύτερο μισό στη συνέχεια, ο DMA γεμίζει διαδοχικά τα δύο μισά του buffer· όταν ολοκληρωθεί η πλήρωση του πρώτου μισού, δημιουργείται half-transfer event· τέλος, όταν ολοκληρωθεί η πλήρωση και του δεύτερου μισού, δημιουργείται transfer-complete event. Η τεχνική αυτή επιτρέπει στον DMA να συνεχίζει τη δειγματοληψία, ενώ ταυτόχρονα ο CPU επεξεργάζεται δεδομένα που έχουν ήδη ολοκληρωθεί. Με τον τρόπο αυτό επιτυγχάνεται πραγματική παράλληλη λειτουργία συλλογής και επεξεργασίας.

Το double buffering χρησιμοποιείται ευρέως σε audio systems, video processing, software defined radio, και acquisition εφαρμογές υψηλού throughput. Στην παρούσα εργασία χρησιμοποιείται ώστε: να πραγματοποιείται συνεχής συλλογή δεδομένων, ενώ παράλληλα γίνεται πακετοποίηση και μετάδοση USB. Χωρίς double buffering, ο επεξεργαστής θα έπρεπε να σταματά προσωρινά τη συλλογή κατά την επεξεργασία των δεδομένων, κάτι που θα οδηγούσε σε απώλειες δειγμάτων.

2.3.5 Interrupt-driven processing

Παρότι το DMA μειώνει δραστικά τη χρήση interrupts, εξακολουθεί να απαιτείται μηχανισμός ενημέρωσης του firmware όταν νέα δεδομένα είναι διαθέσιμα. Για τον σκοπό αυτό χρησιμοποιούνται DMA interrupts. Τα σημαντικότερα DMA interrupts σε acquisition εφαρμογές είναι η half-transfer interrupt και η και transfer-complete interrupt. Το half-transfer interrupt ενεργοποιείται όταν γεμίσει το πρώτο μισό του buffer. Το transfer-complete interrupt ενεργοποιείται όταν ολοκληρωθεί η πλήρωση ολόκληρου του buffer.

Η στρατηγική αυτή επιτρέπει interrupt-driven processing σε επίπεδο blocks δεδομένων αντί για μεμονωμένα samples. Η block-based επεξεργασία είναι πολύ πιο αποδοτική διότι μειώνει δραστικά τη συχνότητα interrupts, βελτιώνει τη χρήση cache, και μειώνει το φόρτο συγχρονισμού (synchronization overhead). Επιπλέον, η επεξεργασία μεγάλων blocks είναι πιο αποδοτική για USB packetization, ψηφιακά φίλτρα, επεξεργασία FFT, και αποθήκευση δεδομένων. Τέλος, η interrupt-driven προσέγγιση παρέχει επίσης ντετερμινιστική συμπεριφορά, καθώς ο επεξεργαστής ενεργοποιείται μόνο όταν υπάρχει ουσιαστική ποσότητα νέων δεδομένων προς επεξεργασία.

2.3.6 Memory throughput

Σε συστήματα συλλογής υψηλής ταχύτητας, ο ρυθμός μεταφοράς δεδομένων στη μνήμη αποτελεί κρίσιμο παράγοντα απόδοσης. Το συνολικό memory throughput εξαρτάται από τον ρυθμό παραγωγής δεδομένων, το μέγεθος των δειγμάτων, τον αριθμό ροών DMA, και την αρχιτεκτονική μνήμης του μικροελεγκτή. Στην παρούσα εργασία, για τρία κανάλια 16-bit και sample rate 1 MSPS ανά κανάλι, ο θεωρητικός ρυθμός δεδομένων είναι:

$$R = 3 \times 10^6 \times 16 = 48 \text{ Mbps}$$

Ο όγκος αυτός δεδομένων πρέπει να μεταφερθεί αρχικά από τους ADC προς τη RAM, και στη συνέχεια προς το USB subsystem.

Η ταυτόχρονη λειτουργία πολλών ροών DMA δημιουργεί σημαντικό memory traffic. Ταυτόχρονα, ο CPU και το περιφερειακό USB προσπελαύνουν επίσης τη RAM. Η κατάσταση αυτή δημιουργεί (συμφόρηση διαύλου) bus contention, δηλαδή ανταγωνισμό μεταξύ διαφορετικών hardware masters για πρόσβαση στη μνήμη.

Ο STM32H743ZI διαθέτει σύνθετη αρχιτεκτονική μνήμης με:

- AXI bus,
- πολλαπλά SRAM domains,
- cache μνήμη,
- και διαφορετικές περιοχές μνήμης με διαφορετικά χαρακτηριστικά απόδοσης.

Η σωστή επιλογή memory region για τους DMA buffers επηρεάζει σημαντικά την εφικτή απόδοση (achievable throughput), τη καθυστέρηση, και τη σταθερότητα του pipeline συλλογής.

2.3.7 Cache και coherency προβλήματα

Η χρήση cache στον πυρήνα Cortex-M7 βελτιώνει σημαντικά την απόδοση του επεξεργαστή, αλλά δημιουργεί προβλήματα όταν χρησιμοποιείται DMA, καθώς ο τελευταίος προσπελαύνει απευθείας τη φυσική RAM χωρίς να ενημερώνει την cache του CPU. Ως αποτέλεσμα, ο επεξεργαστής μπορεί να ‘βλέπει’ παλιά δεδομένα που παραμένουν cached, ενώ ο DMA έχει ήδη γράψει νέα δεδομένα στη RAM.

Το πρόβλημα αυτό ονομάζεται cache coherency problem και είναι ιδιαίτερα σημαντικό σε συστήματα συλλογής υψηλής ταχύτητας. Για την αποφυγή του προβλήματος απαιτείται:

- invalidate της cache μετά από DMA writes,
- σωστό memory alignment,
- ή χρήση non-cacheable περιοχών της μνήμης.

Η σωστή διαχείριση της cache είναι κρίσιμη ώστε τα δεδομένα acquisition να παραμένουν αξιόπιστα.

2.3.8 DMA και πραγματικός χρόνος

Η χρήση DMA βελτιώνει σημαντικά τη real-time συμπεριφορά του συστήματος. Επειδή οι μεταφορές πραγματοποιούνται ανεξάρτητα από τον CPU, η χρονική συμπεριφορά του acquisition subsystem γίνεται πιο ντετερμινιστική.

Ο επεξεργαστής δεν χρειάζεται να εξυπηρετεί κάθε sample ξεχωριστά και έτσι μειώνεται το interrupt jitter, σταθεροποιείται το timing, και αυξάνεται η αξιοπιστία του acquisition pipeline. Το DMA αποτελεί βασικό εργαλείο σχεδίασης συστημάτων πραγματικού χρόνου, ιδιαίτερα όταν συνδυάζεται με:

- hardware triggering,
- κυκλικό buffering,

- και interrupt-driven block processing.

2.3.9 Συμπεράσματα

Το DMA αποτελεί θεμελιώδες στοιχείο σύγχρονων ενσωματωμένων συστημάτων συλλογής και επιτρέπει τη διαχείριση μεγάλου όγκου δεδομένων με ελάχιστη επιβάρυνση του επεξεργαστή. Η χρήση κυκλικής λειτουργίας και double buffering επιτρέπει συνεχή συλλογή δεδομένων χωρίς διακοπές, ενώ η οδηγούμενη-από-interrupt επεξεργασία ανά block προσφέρει αποδοτική και ντετερμινιστική λειτουργία. Παράλληλα, το memory throughput (ταχύτητα μεταφοράς δεδομένων μνήμης) και η σωστή διαχείριση cache αποτελούν κρίσιμους παράγοντες για την επίτευξη υψηλής απόδοσης σε συστήματα βασισμένα στον STM32H743ZI και στον πυρήνα ARM Cortex-M7.

2.4 Τεχνολογία USB

Η τεχνολογία USB (Universal Serial Bus) αποτελεί σήμερα το επικρατέστερο σύστημα διασύνδεσης περιφερειακών συσκευών με ηλεκτρονικούς υπολογιστές και ενσωματωμένα συστήματα. Από την πρώτη εμφάνισή του στα μέσα της δεκαετίας του 1990, το USB σχεδιάστηκε με στόχο να παρέχουν έναν ενιαίο και εύχρηστο τρόπο επικοινωνίας μεταξύ συστημάτων host και περιφερειακών συσκευών, αντικαθιστώντας τις παλαιότερες σειριακές και παράλληλες διεπαφές.

Η μεγάλη εξάπλωση του USB οφείλεται σε μια σειρά σημαντικών χαρακτηριστικών, όπως:

- η αυτόματη αναγνώριση συσκευών (plug and play),
- η δυνατότητα σύνδεσης εν ενεργεία (hot-plugging),
- η τυποποιημένη παροχή τροφοδοσίας,
- η υψηλή αξιοπιστία επικοινωνίας,
- και η υποστήριξη μεγάλου εύρους ταχυτήτων μεταφοράς δεδομένων.

Στα σύγχρονα ενσωματωμένα συστήματα, το USB χρησιμοποιείται εκτενώς για:

- τη μεταφορά δεδομένων,
- την επικοινωνία με υπολογιστές,
- τις ενημερώσεις (updates) του firmware,
- τον εντοπισμό και αποκατάσταση σφαλμάτων (debugging),
- και τις εφαρμογές συνεχούς ροής (streaming).

Στην παρούσα εργασία το USB χρησιμοποιείται ως κύριο μέσο μεταφοράς δεδομένων μεταξύ του μικροελεγκτή STM32H743ZI και του υπολογιστή. Μέσω της διεπαφής αυτής μεταφέρονται οι μετρήσεις των μετατροπέων ADC προς την εφαρμογή επεξεργασίας που αναπτύχθηκε σε Python.

Η χρήση USB σε εφαρμογές συλλογής δεδομένων υψηλής ταχύτητας συνδυάζει σημαντικά πλεονεκτήματα αλλά και σοβαρούς περιορισμούς ως προς τη διεκπεραιωτικότητα. Η κατανόηση των παραπάνω είναι επομένως απαραίτητη για τη σωστή σχεδίαση του συστήματος. Για τον λόγο αυτό θα επιχειρήσω στη συνέχεια να περιγράψω καταρχήν την αρχιτεκτονική και λειτουργία του USB.

2.4.1 Αρχιτεκτονική των USB

Η αρχιτεκτονική των USB βασίζεται σε ένα μοντέλο κεντρικής συσκευής (host-device). Σε αντίθεση με άλλες τεχνολογίες επικοινωνίας, όπου δύο συσκευές μπορούν να επικοινωνούν ‘συμμετρικά’, στο USB υπάρχει πάντα ένας κεντρικός ελεγκτής (host controller), ο οποίος ελέγχει πλήρως την επικοινωνία.

Συνήθως ο host είναι ένας προσωπικός υπολογιστής, ενώ οι υπόλοιπες συσκευές λειτουργούν ως USB. Η επικοινωνία πραγματοποιείται αποκλειστικά με πρωτοβουλία του host, ο οποίος αναλαμβάνει:

- την ανίχνευση τυχόν νέων συσκευών,
- την ανάθεση διευθύνσεων,
- τη διαχείριση του εύρους ζώνης,
- και τον προγραμματισμό των μεταφορών δεδομένων.

Η USB συσκευή δεν μπορεί να ξεκινήσει αυθαίρετα επικοινωνία· αντίθετα, απαντά μόνο σε αιτήματα που αποστέλλονται από τον host. Η φιλοσοφία αυτή επιτρέπει την απλοποίηση των peripheral devices, αλλά ταυτόχρονα δημιουργεί περιορισμούς σε εφαρμογές συνεχούς ροής, καθώς ο host καθορίζει πότε και πόσο συχνά θα πραγματοποιούνται οι μεταφορές δεδομένων.

Στην περίπτωση της παρούσας εργασίας, ο μικροελεγκτής λειτουργεί ως USB και ο υπολογιστής ως USB host. Τα δεδομένα που συλλέγονται μεταφέρονται με μαζικές μεταφορές (bulk transfers) από το ενσωματωμένο σύστημα προς την εφαρμογή του υπολογιστή.

2.4.2 Εκδόσεις USB και ταχύτητες

Το πρότυπο USB έχει εξελιχθεί σημαντικά με την πάροδο των ετών. Οι βασικές εκδόσεις περιλαμβάνουν:

- USB 1.1,
- USB 2.0,
- USB 3.x.

Η παρούσα εργασία βασίζεται σε Full Speed USB, το οποίο αποτελεί μέρος του προτύπου USB 1.1 και υποστηρίζει θεωρητικό μέγιστο ρυθμό μετάδοσης:

$$R_{USB} = 12\ Mbps$$

Η θεωρητική αυτή τιμή όμως δεν είναι πρακτικά διαθέσιμη στο application layer. Στην πράξη σημαντικό μέρος του εύρους ζώνης καταναλώνεται από:

- επικεφαλίδες πακέτων (packet headers),
- πεδία συγχρονισμού (synchronization fields),
- CRC,
- καθυστερήσεις μεταξύ πακέτων (inter-packet delays),
- και επιβάρυνση του χρονοπρογραμματισμού κεντρικού υπολογιστή (host scheduling overhead).

Ως αποτέλεσμα, ο πραγματικός βαθμός διεκπεραιωτικότητας για τις μαζικές μεταφορές σε ένα USB Full Speed είναι σημαντικά μικρότερος και συνήθως κυμαίνεται περίπου μεταξύ 800 kB/s και 1 MB/s.

Ο περιορισμός αυτός αποτελεί ένα από τα σημαντικότερα τεχνικά ζητήματα που επιχείρησα να αντιμετωπίσω στο πλαίσιο της παρούσας εργασίας, καθώς ο παραγόμενος όγκος δεδομένων από τους μετατροπείς ADC υπερβαίνει κατά πολύ τις δυνατότητες συνεχούς μεταφοράς σε πραγματικό χρόνο που έχει ένα Full Speed USB.

2.4.3 USB Frames και χρονισμός

Η επικοινωνία USB οργανώνεται χρονικά στα λεγόμενα frames. Στο USB Full Speed κάθε frame έχει διάρκεια (T_{frame}) $1\,ms$. Κατά τη διάρκεια του κάθε frame ο host προγραμματίζει τις μεταφορές δεδομένων όλων των συνδεδεμένων συσκευών. Το γεγονός αυτό σημαίνει ότι το διαθέσιμο εύρος ζώνης μοιράζεται δυναμικά μεταξύ όλων των ενεργών συσκευών USB.

Η λειτουργία του USB που βασίζεται στα frames επηρεάζει άμεσα:

- τον χρόνο απόκρισης (latency),
- την ικανότητα διεκπεραίωσης,
- και τη ντετερμινιστική συμπεριφορά της επικοινωνίας.

Σε εφαρμογές συλλογής δεδομένων υψηλής ταχύτητας ο προγραμματισμός του host αποτελεί σημαντικό παράγοντα απόδοσης, καθώς μικρές καθυστερήσεις μπορούν να οδηγήσουν σε προσωρινή συσσώρευση δεδομένων στους buffers του μικροελεγκτή.

Για τον λόγο αυτό απαιτείται σωστή στρατηγική buffering και ‘πακετοποίησης’ ώστε να αποφεύγονται οι υπερχειλίσεις δεδομένων.

2.4.4 USB Endpoints

Η επικοινωνία USB βασίζεται στην έννοια των τελικών σημείων (endpoints). Ένα τελικό σημείο αποτελεί το λογικό κανάλι επικοινωνίας μεταξύ host και της συσκευής.

Κάθε συσκευή USB διαθέτει ένα τελικό σημείο 0, το οποίο χρησιμοποιείται για την απαρίθμηση (enumeration), την παραμετροποίηση (configuration), και τις μεταφορές ελέγχου (control transfers). Πέρα από το endpoint 0, η συσκευή μπορεί να διαθέτει επιπλέον τελικά σημεία διαφορετικών τύπων ανάλογα με τις απαιτήσεις της εφαρμογής.

Τα endpoints διακρίνονται σε σημεία: ελέγχου (control), μαζικής επεξεργασίας (bulk), διακοπής (interrupt) και ισόχρονα (isochronous).

Στην παρούσα εργασία χρησιμοποιούνται τελικά σημεία μαζικής επεξεργασίας, καθώς είναι κατάλληλα για αξιόπιστη μεταφορά δεδομένων, έλεγχο ενδεχόμενων λαθών μέσω CRC και μέγιστη αξιοποίηση του διαθέσιμου εύρους ζώνης.

Τα bulk transfers είναι ιδιαίτερα κατάλληλα για εφαρμογές μεταφοράς μεγάλου όγκου δεδομένων όπου η αξιοπιστία είναι σημαντικότερη από τη deterministic latency.

2.4.5 Μαζικές μεταφορές (Bulk Transfers)

Οι μαζικές μεταφορές (bulk transfers) χρησιμοποιούνται κυρίως όταν πρόκειται για συνεχείς ροές δεδομένων, μεταφορά αρχείων, συστήματα συλλογής δεδομένων και γενικά για εφαρμογές με ανάγκες διεκπεραίωσης μεγάλου όγκου εργασιών.

Το πρωτόκολλο USB εγγυάται ότι τα πακέτα μαζικών μεταφορών θα παραδοθούν σωστά μέσω:

- CRC checking,
- ACK/NACK μηχανισμών,
- και επαναμετάδοση σε περίπτωση σφάλματος.

Το χαρακτηριστικό αυτό είναι ιδιαίτερα σημαντικό για εφαρμογές συλλογής δεδομένων, όπου ελλοχεύει ο κίνδυνος αλλοίωσης των δεδομένων κατά τη μεταφορά.

Ωστόσο, τα μαζικές μεταφορές δεν παρέχουν εγγυημένη απόκριση ή εύρος ζώνης, καθώς ο host μόνο αφού εξυπηρετήσει τις μεταφορές υψηλότερης προτεραιότητας, όπως τα ισόχρονα (isochronous) ή τα interrupts, θα διαθέσει χρόνο σε τελικά σημεία μεταφοράς. Η προτεραιοποίηση αυτή των εργασιών εκ μέρους του host σημαίνει πρακτικά ότι η 'ακαριαία' διεκπεραίωση μπορεί να παρουσιάσει διακυμάνσεις και, επομένως, ότι απαιτείται buffering στο firmware του μικροελεγκτή ώστε να απορροφώνται τυχόν προσωρινές καθυστερήσεις της USB επικοινωνίας.

2.4.6 Enumeration διαδικασία

Όταν μια USB συσκευή συνδέεται στον υπολογιστή, πραγματοποιείται μια διαδικασία αναγνώρισης γνωστή ως enumeration.

Κατά τη διαδικασία αυτή:

1. Ο host ανιχνεύει τη νέα συσκευή.
2. Αναθέτει προσωρινή διεύθυνση.
3. Ζητά τους περιγραφείς (descriptors)[3] της συσκευής.
4. Αναγνωρίζει τον τύπο της συσκευής.
5. Φορτώνει τον κατάλληλο driver.
6. Ενεργοποιεί τα απαραίτητα τελικά σημεία.

Στην αρχική υλοποίηση της παρούσας εργασίας χρησιμοποιήθηκε CDC κλάση USB (βλ. παρακάτω), η οποία αναγνωρίζεται αυτόματα ως εικονική σειριακή θύρα (virtual serial port). Στη συνέχεια χρησιμοποιήθηκε ειδική κλάση (custom class) USB με περιγραφείς κατασκευαστή και τελικά σημεία μαζικής επεξεργασίας ώστε να επιτευχθεί μεγαλύτερη διεκπεραίωση εργασιών και καλύτερος έλεγχος της επικοινωνίας.

2.4.7 Κλάσεις USB (USB Classes)

Το πρότυπο USB ορίζει διαφορετικές προτυποποιημένες κλάσεις ώστε οι συσκευές να αναγνωρίζονται αυτόματα από το λειτουργικό σύστημα. Μερικές από τις πιο διαδεδομένες κλάσεις είναι οι HID, Mass Storage, Audio, CDC και Video class. Η CDC (Communications Device Class) χρησιμοποιείται ευρέως για virtual serial ports. Το βασικό της πλεονέκτημα είναι ότι υποστηρίζεται εγγενώς από τα περισσότερα λειτουργικά συστήματα χωρίς ανάγκη για custom drivers. Ωστόσο, η CDC είναι γνωστό ότι δημιουργεί επιπλέον φόρτο, επειδή προσομοιώνει σειριακή επικοινωνία UART πάνω από USB (περιορίζοντας το βαθμό διεκπεραιωτικότητας και αυξάνοντας τον χρόνο απόκρισης). Για εφαρμογές συλλογής δεδομένων υψηλού εύρους ζώνης, όπως στην παρούσα εργασία, προτιμάται συνήθως ειδική κλάση

[3] Οι πληροφορίες που προσφέρουν οι περιγραφείς περιλαμβάνουν: τον κατασκευαστή, το VID/PID, τον αριθμό endpoints, και τον τύπο της USB class.

(custom class) USB με τελικά σημεία μαζικής επεξεργασίας, καθώς παρέχει χαμηλότερο φόρτο, μεταφορές πακέτων και καλύτερη αξιοποίηση του εύρους ζώνης του USB.

2.4.8 Drivers και επικοινωνία με το λειτουργικό σύστημα

Η επικοινωνία μιας USB συσκευής με το λογισμικό του υπολογιστή απαιτεί την ύπαρξη κατάλληλου driver. Στα συστήματα Linux είναι ιδιαίτερα διαδεδομένη η χρήση της βιβλιοθήκης libusb, η οποία επιτρέπει από τον χώρο του χρήστη πρόσβαση σε συσκευές USB χωρίς ανάπτυξη κάποιου custom kernel driver. Συγκεκριμένα, στα Windows υπάρχουν δύο βασικές προσεγγίσεις:

- ανάπτυξη custom kernel-mode driver,
- ή χρήση generic user-mode drivers όπως το WinUSB.

Η ανάπτυξη kernel driver παρέχει μεγαλύτερη ευελιξία αλλά συνοδεύεται από σημαντική πολυπλοκότητα και απαιτήσεις ψηφιακής υπογραφής του driver. Για τον λόγο αυτό στην παρούσα εργασία επιλέχθηκε η χρήση WinUSB, το οποίο επιτρέπει user-space πρόσβαση στα bulk endpoints μέσω εφαρμογής Python.

Η προσέγγιση αυτή μειώνει σημαντικά την πολυπλοκότητα ανάπτυξης και επιτρέπει την ταχύτερη δημιουργία acquisition software.

2.4.9 USB και ενσωματωμένα συστήματα συλλογής δεδομένων

Η χρήση των USB στα ενσωματωμένα συστήματα συλλογής δεδομένων παρουσιάζει σημαντικά πλεονεκτήματα. Το USB, λόγω και της ευρείας διάδοσής του, διαθέτει ευρεία υποστήριξη από λειτουργικά συστήματα και παρέχει ικανοποιητικούς ρυθμούς μεταφοράς για μεγάλο αριθμό εφαρμογών. Ωστόσο, όπως αναφέραμε προηγουμένως, σε εφαρμογές υψηλής ταχύτητας εμφανίζει σημαντικούς περιορισμούς. Στην παρούσα εργασία ο συνολικός ρυθμός δεδομένων των μετατροπέων ADC είναι:

$$R = 48\,Mbps$$

Ένας ρυθμός δηλαδή κατά πολύ μεγαλύτερος από τις δυνατότητες του Full Speed USB.

Η παρατήρηση αυτή επηρέασε καθοριστικά τη σχεδίαση του συστήματος και οδήγησε στην υλοποίηση: στρατηγικών buffering, 'πακετοποίηση' βασισμένη σε DMA και διαφορετικών τρόπων λειτουργίας.

Σε χαμηλότερους ρυθμούς δειγματοληψίας το USB χρησιμοποιείται για συνεχή ροή, ενώ για ακόμη υψηλότερους ρυθμούς γίνεται πρώτα προσωρινή αποθήκευση στη RAM και σε δεύτερο χρόνο αποστολή στον υπολογιστή.

2.4.10 Συμπεράσματα

Η τεχνολογία USB αποτελεί βασικό στοιχείο των σύγχρονων ενσωματωμένων συστημάτων και παρέχει έναν ευέλικτο μηχανισμό επικοινωνίας μεταξύ μικροελεγκτών και υπολογιστών που υποστηρίζεται ευρύτατα σήμερα.

Παρά τα σημαντικά πλεονεκτήματα της τεχνολογίας, η χρήση της σε εφαρμογές συλλογής δεδομένων υψηλής διεκπεραιωτικότητας παρουσιάζει περιορισμούς. Αυτοί σχετίζονται κυρίως με το διαθέσιμο εύρος ζώνης και το μοντέλο της επικοινωνίας με τον κεντρικό υπολογιστή.

Η κατανόηση της αρχιτεκτονικής του USB που επιχειρήσαμε σε αυτή την ενότητα, των μαζικών μεταφορών, των τελικών σημείων (endpoints) και της διαδικασίας enumeration, που είναι απαραίτητη

για τη σωστή σχεδίαση firmware και host software, στην παρούσα εργασία αξιοποιούνται με σκοπό την αξιόπιστη μεταφορά δεδομένων μεταξύ του συστήματος συλλογής δεδομένων και της εφαρμογής επεξεργασίας σε Python.

Κεφάλαιο 3ο: Σχεδίαση Συστήματος

3.1 Hardware Architecture

Η σχεδίαση του hardware αποτελεί εξαιρετικά σημαντικό στάδιο της σχεδίασης ενός συστήματος συλλογής δεδομένων υψηλής ταχύτητας, καθώς καθορίζει άμεσα: (α) τις δυνατότητες δειγματοληψίας, (β) την αξιοπιστία λειτουργίας και (γ) τη συνολική απόδοση του συστήματος. Η συνολική αρχιτεκτονική του πολυκαναλικού συστήματος, το οποίο σχεδιάστηκε στο πλαίσιο της παρούσας εργασίας, έχοντας σκοπό τη συλλογή και μεταφορά αναλογικών δεδομένων υψηλής ταχύτητας προς ηλεκτρονικό υπολογιστή μέσω διασύνδεσης USB και αξιοποιώντας τον μικροελεγκτή STM32H743ZI, είχε βασικό γνώμονα:

- την επίτευξη υψηλού ρυθμού δειγματοληψίας,
- τη μείωση της επιβάρυνσης του CPU
- την αξιόπιστη μεταφορά δεδομένων,
- και τη δυνατότητα συνεχούς λειτουργίας χωρίς απώλειες δειγμάτων.

Το υποσύστημα του hardware αποτελείται από:

- το αναλογικό τμήμα εισόδου,
- τους ADC του μικροελεγκτή,
- το υποσύστημα DMA,
- τη μνήμη RAM,
- τη διασύνδεση USB,
- και τα κυκλώματα χρονισμού και τροφοδοσίας.

Η συνεργασία όλων των παραπάνω υποσυστημάτων επιτρέπει τη δημιουργία ενός ολοκληρωμένου acquisition pipeline , στο οποίο το γενικό σκεπτικό είναι ότι τα αναλογικά σήματα, αφού μετατραπούν σε ψηφιακά δείγματα, αποθηκεύονται προσωρινά στη μνήμη και στη συνέχεια μεταφέρονται στον υπολογιστή για επεξεργασία.

3.1.1 Επιλογή μικροελεγκτή

Ο μικροελεγκτής που χρησιμοποιήθηκε στην παρούσα εργασία είναι ο STM32H743ZI της εταιρείας STMicroelectronics. Η επιλογή του συγκεκριμένου μικροελεγκτή πραγματοποιήθηκε λαμβάνοντας υπόψη τις αυξημένες δυνατότητες που παρουσιάζει σε εφαρμογές υψηλής απόδοσης και κατά τη συλλογή δεδομένων σε πραγματικό χρόνο.

Ο μικροελεγκτής αυτός βασίζεται στον πυρήνα ARM Cortex-M7 και μπορεί να λειτουργήσει σε συχνότητες έως 480 MHz. Αυτή η υψηλή υπολογιστική ισχύς επιτρέπει την ταυτόχρονη λειτουργία πολλαπλών υποσυστημάτων χωρίς να επιβαρύνεται σημαντικά ο επεξεργαστής. Παράλληλα, ο συγκεκριμένος μικροελεγκτής διαθέτει μεγάλη ποσότητα ενσωματωμένης SRAM, πολλαπλούς ελεγκτές DMA και προηγμένα περιφερειακά επικοινωνίας.

Ένας ακόμη από τους βασικούς λόγους επιλογής του συγκεκριμένου μικροελεγκτή υπήρξε το γεγονός ότι διαθέτει τρεις ανεξάρτητους μετατροπείς ADC υψηλής ταχύτητας. Οι ADC αυτοί μπορούν να

λειτουργήσουν και ανεξάρτητα αλλά και σε συνδυασμό με άλλες λειτουργίες, επιτρέποντας τη δημιουργία πολυκαναλικών συστημάτων συλλογής δεδομένων με συγχρονισμένη δειγματοληψία.

Επιπλέον, ο μικροελεγκτής STM32H743ZI διαθέτει ενσωματωμένο ένα USB OTG Full Speed peripheral, γεγονός που επιτρέπει την απευθείας σύνδεση με ηλεκτρονικό υπολογιστή χωρίς ανάγκη εξωτερικού ελεγκτή USB.

Τέλος, η συγκεκριμένη επιλογή μικροελεγκτή παρέχει σημαντική ευελιξία στη διαχείριση μεταφορών DMA και στρατηγικών buffering – κάτι αναμφίβολα σημαντικό σε εφαρμογές συλλογής υψηλής διεκπεραιωτικότητας – καθώς προσφέρει πολλαπλούς τομείς μνήμης, προσωρινή μνήμη αλλά και διαφορετικών διαύλους πρόσβασης στη RAM.

3.1.2 Συνολική αρχιτεκτονική συστήματος

Η συνολική λειτουργία του συστήματος μπορεί να διακριθεί σε τέσσερα βασικά στάδια:

1. το στάδιο της λήψης των αναλογικών σημάτων,
2. το στάδιο της μετατροπής σε ψηφιακή μορφή,
3. το στάδιο της προσωρινής αποθήκευσης στη μνήμη,
4. Και, τέλος, το στάδιο της μεταφοράς δεδομένων στον υπολογιστή μέσω USB.

Τα αναλογικά σήματα εισόδου οδηγούνται απευθείας στα κανάλια ADC του μικροελεγκτή. Οι μετατροπείς ADC εκτελούν συνεχείς μετατροπές με προκαθορισμένο ρυθμό δειγματοληψίας και τα αποτελέσματα μεταφέρονται αυτόματα στη RAM μέσω DMA[4] χωρίς συνεχή παρέμβαση του επεξεργαστή. Στη συνέχεια, τα δεδομένα αποθηκεύονται σε διπλούς κυκλικούς buffers, επιτρέποντας τη συνεχή λειτουργία του υποσυστήματος συλλογής. Ενώ, δηλαδή, το DMA γεμίζει το ένα μισό του buffer, το firmware μπορεί να επεξεργάζεται ή να μεταδίδει το άλλο μισό προς τον υπολογιστή.

Η επικοινωνία με τον υπολογιστή πραγματοποιείται μέσω ενός Full Speed USB. Το firmware οργανώνει τα δεδομένα σε πακέτα και τα αποστέλλει μέσω των τελικών σημείων μαζικής επεξεργασίας προς την εφαρμογή λήψης σε Python.

Με όλα τα παραπάνω επιτυγχάνεται παράλληλη λειτουργία συλλογής δεδομένων και επικοινωνίας, αποδοτική χρήση της RAM, και ελαχιστοποίηση των καθυστερήσεων.

3.1.3 Αναλογικό υποσύστημα εισόδου

Το αναλογικό υποσύστημα αποτελεί το σημείο εισόδου των σημάτων προς μέτρηση. Πρέπει εδώ να σημειωθεί ότι η ποιότητα του αναλογικού front-end επηρεάζει άμεσα την ακρίβεια και τη σταθερότητα των μετρήσεων. Οι είσοδοι των μετατροπέων ADC του μικροελεγκτή λειτουργούν σε συγκεκριμένο εύρος τάσης, το οποίο καθορίζεται από την τάση αναφοράς και τα χαρακτηριστικά του μικροελεγκτή. Για τον λόγο αυτό τα αναλογικά σήματα πρέπει να παραμένουν εντός των επιτρεπτών ορίων λειτουργίας. Επίσης, επειδή έχει σημασία η αντίσταση εξόδου της πηγής σήματος και οι SAR ADC διαθέτουν εσωτερικό πυκνωτή δειγματοληψίας και συγκράτησης, αυτός πρέπει να φορτίζεται σωστά κατά τη διάρκεια της δειγματοληψίας. Σε άλλη περίπτωση, λόγω ατελούς φόρτισης του sample

[4] Η χρήση των ελεγκτών DMA είναι κρίσιμη για την επίτευξη υψηλών ρυθμών συλλογής, καθώς μειώνει δραστικά τον φόρτο του CPU. Αντί ο επεξεργαστής να διαβάζει κάθε δείγμα ξεχωριστά, οι DMA μεταφέρουν μεγάλα μπλοκ δεδομένων απευθείας στη μνήμη.

capacitor, η πηγή παρουσιάζει υψηλή αντίσταση εξόδου και ενδέχεται να εμφανιστούν σφάλματα μετατροπής. Τέλος, σε εφαρμογές υψηλής ταχύτητας είναι σημαντική η αποφυγή ηλεκτρικού θορύβου[5] και παρεμβολών. Για τον λόγο αυτό απαιτείται προσεκτική σχεδίαση του PCB και σωστός διαχωρισμός αναλογικών και ψηφιακών γειώσεων.

3.1.4 Υποσύστημα ADC

Ο μικροελεγκτής STM32H743ZI διαθέτει τρεις ανεξάρτητους μετατροπείς ADC υψηλής ταχύτητας. Στην παρούσα εργασία χρησιμοποιούνται και οι τρεις ώστε να επιτευχθεί πολυκαναλική δειγματοληψία. Συγκεκριμένα, οι ADC1 και ADC2 βρίσκονται σε διπλή ταυτόχρονη λειτουργία, ενεργοποιούνται ταυτόχρονα μέσω κοινού hardware trigger, επιτρέποντας με αυτό τον τρόπο συγχρονισμένη δειγματοληψία δύο καναλιών. Η δυνατότητα αυτή είναι ιδιαίτερα σημαντική σε εφαρμογές όπου απαιτείται μικρό χρονικό απόκλιση μεταξύ των σημάτων. Ο ADC3 λειτουργεί ανεξάρτητα, παρέχοντας τρίτο κανάλι συλλογής. Επιλέχθηκε χρήση ξεχωριστού ADC επειδή έτσι επιτυγχάνεται μεγαλύτερη ευελιξία στη διαμόρφωση του συστήματος και καλύτερη κατανομή των resources της DMA.

Η ενεργοποίηση των ADC πραγματοποιείται μέσω χρονιστών hardware ώστε να εξασφαλίζεται σταθερός και ντετερμινιστικός χρονισμός δειγματοληψίας ανεξάρτητα από το χρόνο απόκρισης του ή τις τυχόν διακοπές. Η χρήση σκανδάλης για το hardware είναι ιδιαίτερα σημαντική σε εφαρμογές συλλογής υψηλής ταχύτητας, καθώς επιτρέπει ακριβή χρονικό συγχρονισμό και μειώνει το jitter μεταξύ διαδοχικών μετατροπών.

3.1.5 Υποσύστημα DMA

Η χρήση των ελεγκτών DMA αποτελεί θεμελιώδες στοιχείο της αρχιτεκτονικής του συστήματος. Χωρίς αυτούς θα ήταν πρακτικά αδύνατη η συνεχής συλλογή δεδομένων σε ρυθμούς της τάξης του 1 MSPS ανά κανάλι. Στην παρούσα υλοποίηση χρησιμοποιούνται DMA1, DMA2, και BDMA, για την εξυπηρέτηση των τριών ADC. Οι ελεγκτές των DMA μεταφέρουν αυτόματα τα αποτελέσματα μετατροπής από τα ADC data registers προς τη RAM. Η μεταφορά πραγματοποιείται χωρίς άμεση εμπλοκή του CPU, επιτρέποντας στον επεξεργαστή να ασχολείται με την πακετοποίηση, την επικοινωνία με το USB, και τη διαχείριση του buffer. Τα DMA λειτουργούν σε circular double-buffer mode. Η τεχνική αυτή επιτρέπει τη συνεχή επαναχρησιμοποίηση των buffers χωρίς διακοπή της δειγματοληψίας. Η χρήση διακοπών ‘half-transfer’ και ‘transfer-complete’ επιτρέπει στο firmware να ενημερώνεται όταν γεμίζει το μισό ή ολόκληρο το buffer. Με αυτόν τον τρόπο επιτυγχάνεται συνεχής ροή δεδομένων χωρίς ανάγκη διαρκούς polling.

3.1.6 Μνήμη RAM και buffering

Λόγω του μεγάλου όγκου δεδομένων που παράγεται από τους ADC κρίσιμη υπήρξε κατά τη σχεδίαση του συστήματος η επίλυση των ζητημάτων που αφορούσαν τη διαχείριση της μνήμης. Ο μικροελεγκτής διαθέτει πολλαπλές περιοχές SRAM με διαφορετικά χαρακτηριστικά πρόσβασης. Η σωστή επιλογή περιοχής μνήμης για τα buffers της μνήμης DMA επηρεάζει όχι μόνο τη διεκπεραιωτικότητα αλλά και τη συμβατότητα με τη DMA και τη λειτουργία της προσωρινής μνήμης (cache).

[5] Ο θόρυβος μπορεί να προέρχεται από την τροφοδοσία, από ψηφιακά switching signals, από το USB interface, ή από εξωτερικές ηλεκτρομαγνητικές παρεμβολές.

Ιδιαίτερη προσοχή απαίτησε και η data cache του πυρήνα Cortex-M7 καθώς οι ελεγκτές της DMA προσπελαύνουν απευθείας τη RAM χωρίς να ενημερώνουν την cache του επεξεργαστή. Ως αποτέλεσμα, μπορεί να εμφανιστούν προβλήματα αξιοπιστίας των δεδομένων της cache – και προφανώς του συνολικού pipeline – στην περίπτωση που ο επεξεργαστής διαβάσει παλιά δεδομένα (στην cache) αντί για τα νέα δεδομένα του DMA.

Για την αποφυγή τέτοιων προβλημάτων είναι απολύτως απαραίτητο (α) invalidate της cache μετά από DMA writes, ή (β) χρήση non-cacheable περιοχών μνήμης.

3.1.7 Υποσύστημα USB

Η επικοινωνία με τον υπολογιστή πραγματοποιείται μέσω του ενσωματωμένου περιφερειακού USB Full Speed του μικροελεγκτή.

Αρχικά χρησιμοποιήθηκε USB CDC (Virtual COM Port) ώστε να επιτευχθεί γρήγορη ανάπτυξη και εύκολη αποσφαλμάτωση. Ωστόσο, λόγω περιορισμών στη διεκπεραιωτικότητα, στη συνέχεια υλοποιήθηκε ειδική κλάση (custom class) USB με τελικά σημεία μαζικής επεξεργασίας. Η χρήση τελικών σημείων μαζικής επεξεργασίας επιτρέπει αποδοτικότερη μεταφορά μεγάλου όγκου δεδομένων και χαμηλότερο υπερχείλισης πρωτοκόλλου σε σχέση με το CDC.

Παρά τα πλεονεκτήματα αυτά, πρέπει να σημειωθεί ότι το USB Full Speed παραμένει ο κύριος περιοριστικός παράγοντας του συστήματος, καθώς το θεωρητικό μέγιστο εύρος ζώνης είναι:

$$R_{USB}=12 \text{ Mbps}$$

ένας ρυθμός δηλαδή σημαντικά μικρότερος από τον παραγόμενο ρυθμό δεδομένων των μετατροπέων ADC.

Η παρατήρηση αυτή οδήγησε στην ανάπτυξη διαφορετικών τρόπων λειτουργίας:

- buffered acquisition,
- ή συνεχή ροή (stream – για δειγματοληψίες μειωμένου ρυθμού).

3.1.8 Τροφοδοσία και χρονισμός

Η σταθερή τροφοδοσία είναι ιδιαίτερα σημαντική σε συστήματα συλλογής δεδομένων υψηλής ακρίβειας. Θόρυβος στην τροφοδοσία μπορεί να οδηγήσει σε αυξημένη στάθμη θορύβου και σε υποβάθμιση προφανώς της ποιότητας των μετρήσεων. Για τον λόγο αυτό απαιτείται σωστή αποσύζευξη (decoupling) κοντά στον μικροελεγκτή, στους ADC αλλά και στα αναλογικά κυκλώματα.

Από την άλλη πλευρά, ο χρονισμός του συστήματος επηρεάζει άμεσα τη σταθερότητα του ρυθμού δειγματοληψίας. Οι μετατροπείς ADC και το υποσύστημα του USB βασίζονται σε ρολόγια PLL υψηλής ακρίβειας ώστε να επιτυγχάνεται σταθερή λειτουργία.

Η χρήση χρονιστών hardware για την ενεργοποίηση των ADC εξασφαλίζει ντετερμινιστικό συγχρονισμό και μειώνει την αστάθεια του χρονισμού δειγματοληψίας.

3.1.9 Συμπεράσματα

Η αρχιτεκτονική του hardware του συστήματος σχεδιάστηκε με στόχο τη δημιουργία μιας αποδοτικής πλατφόρμας συλλογής δεδομένων υψηλής ταχύτητας βασισμένου στον STM32H743ZI. Η συνδυασμένη χρήση πολλαπλών ADC, ελεγκτών DMA, κυκλικού buffering και επικοινωνίας μέσω USB επιτρέπει τη συνεχή συλλογή και μεταφορά δεδομένων με ελάχιστη επιβάρυνση του επεξεργαστή.

Η σωστή συνεργασία όλων των υποσυστημάτων είναι απαραίτητη για την επίτευξη αξιόπιστης λειτουργίας, δεδομένων των αυξημένων απαιτήσεων για υψηλή διεκπεραιωτικότητα από τη μια και των περιορισμών της επικοινωνίας με USB Full Speed από την άλλη.

3.2 Υποσύστημα ADC

Το υποσύστημα μετατροπής αναλογικού σε ψηφιακό σήμα αποτελεί τον πυρήνα του συστήματος συλλογής δεδομένων που σχεδιάστηκε και υλοποιήθηκε στο πλαίσιο της παρούσας εργασίας. Η συνολική απόδοση του συστήματος εξαρτάται άμεσα από τις δυνατότητες των ADC, τον τρόπο συγχρονισμού τους, τη σταθερότητα του χρονισμού και την αποδοτική μεταφορά των δεδομένων προς τη μνήμη. Στόχος της σχεδίασης ήταν η υλοποίηση ενός πολυκαναλικού συστήματος δειγματοληψίας υψηλής ταχύτητας, ικανού να πραγματοποιεί συνεχείς μετρήσεις σε τρία αναλογικά κανάλια με όσο το δυνατόν μικρότερη χρονική απόκλιση μεταξύ των μετρήσεων και με ελάχιστη επιβάρυνση του επεξεργαστή. Για τον σκοπό αυτό αξιοποιήθηκαν οι τρεις ενσωματωμένοι ADC του μικροελεγκτή STM32H743ZI. Οι ADC αυτοί υποστηρίζουν υψηλές ταχύτητες δειγματοληψίας, hardware triggering, DMA μεταφορές και πολλαπλούς τρόπους λειτουργίας, γεγονός που τους καθιστά κατάλληλους για εφαρμογές συλλογής δεδομένων πραγματικού χρόνου.

Η συνολική αρχιτεκτονική του υποσυστήματος των ADC σχεδιάστηκε με στόχο:

- την επίτευξη υψηλού ρυθμού δειγματοληψίας,
- τη μείωση της αστάθειας του χρονισμού δειγματοληψίας,
- την αξιόπιστη (συνεχή) λειτουργία,
- και την αποδοτική συνεργασία με το DMA και το υποσύστημα του USB.

3.2.1 Εσωτερικοί μετατροπείς ADC του μικροελεγκτή STM32H743ZI

Ο STM32H743ZI διαθέτει τρεις ανεξάρτητους μετατροπείς αναλογικού σε ψηφιακό σήμα (ADC) τύπου SAR (Successive Approximation Register ADC). Η αρχιτεκτονική SAR προσφέρει καλή ισορροπία μεταξύ ταχύτητας, ακρίβειας, και κατανάλωσης ισχύος.

Οι ADC αυτοί έχουν σχεδιαστεί για εφαρμογές υψηλής ταχύτητας και μπορούν να λειτουργήσουν είτε ανεξάρτητα είτε σε συνδυασμένα modes λειτουργίας. Ο κάθε ADC διαθέτει:

- ανεξάρτητο ακολουθητή (διάταξη ελέγχου αλληλουχίας λειτουργιών, sequencer),
- ρυθμιζόμενο χρόνο δειγματοληψίας,
- δυνατότητα ενεργοποίησης hardware,
- διεπαφή DMA,
- και εσωτερικό κύκλωμα δειγματοληψίας και συγκράτησης.

Η λειτουργία των ADC βασίζεται στη διαδοχική προσέγγιση της τάσης εισόδου μέσω εσωτερικού DAC και συγκριτή. Παρότι η διαδικασία αυτή απαιτεί αρκετούς εσωτερικούς κύκλους ρολογιού για κάθε μετατροπή, επιτρέπει την επίτευξη υψηλής ανάλυσης με σχετικά απλό hardware. Οι ADC του STM32H7 υποστηρίζουν ανάλυση έως 16 bits, ωστόσο η πραγματική effective ανάλυση εξαρτάται από τη συχνότητα λειτουργίας, τον ηλεκτρικό θόρυβο, τη σταθερότητα της τροφοδοσίας και τις συνθήκες λειτουργίας του PCB.

Στην παρούσα εργασία οι ADC χρησιμοποιούνται κυρίως σε λειτουργία συνεχούς συλλογής υψηλής ταχύτητας και όχι σε λειτουργία υψηλής ακρίβειας μετρήσεων DC. Για τον λόγο αυτό δόθηκε μεγαλύτερη έμφαση στον ρυθμό δειγματοληψίας, στον συγχρονισμό και στη σταθερότητα του pipeline της συλλογής.

3.2.2 Πολυκαναλική δειγματοληψία

Η ανάγκη ταυτόχρονης παρακολούθησης πολλών αναλογικών σημάτων αποτελεί βασικό χαρακτηριστικό πολλών σύγχρονων συστημάτων συλλογής δεδομένων, όπως ψηφιακή επεξεργασία σημάτων, συστήματα ελέγχου, ενεργειακές μετρήσεις, και ανάλυση πολυφασικών κυματομορφών, όπου είναι σημαντικό οι μετρήσεις διαφορετικών καναλιών να πραγματοποιούνται όσο το δυνατόν πιο συγχρονισμένα.

Στην παρούσα εργασία χρησιμοποιούνται τρία ανεξάρτητα κανάλια συλλογής. Οι ADC1 και ADC2 λειτουργούν σε dual simultaneous mode, ενώ ο ADC3 λειτουργεί ανεξάρτητα. Η χρήση dual mode επιτρέπει στους δύο ADC να ξεκινούν τη μετατροπή ταυτόχρονα μέσω κοινού σκανδαλισμού του hardware. Με αυτόν τον τρόπο μειώνεται σημαντικά η χρονική απόκλιση μεταξύ των δύο καναλιών και επιτυγχάνεται σχεδόν ταυτόχρονη δειγματοληψία. Η δυνατότητα αυτή είναι ιδιαίτερα σημαντική σε εφαρμογές όπου ενδιαφέρει η διαφορά φάσης μεταξύ των σημάτων, η σύγκριση χρονικών μεταβολών, ή η ψηφιακή επεξεργασία πολλών συγχρονισμένων καναλιών. Ο τρίτος ADC λειτουργεί ανεξάρτητα, γεγονός που παρέχει μεγαλύτερη ευελιξία αλλά εισάγει μικρή χρονική διαφορά σε σχέση με τα άλλα δύο κανάλια λόγω της διαφορετικής εσωτερικής λειτουργίας του hardware.

3.2.3 Dual simultaneous mode

Οι ADC1 και ADC2 του STM32H7 υποστηρίζουν πολλαπλά multimode configurations. Στην παρούσα εργασία χρησιμοποιήθηκε λειτουργία διπλού συγχρονισμού (dual simultaneous mode), στο οποίο οι δύο ADC πραγματοποιούν μετατροπή ταυτόχρονα: η έναρξη μετατροπής πραγματοποιείται συγχρονισμένα, και τα αποτελέσματα μπορούν να συνδυαστούν σε κοινό data register. Το dual mode μειώνει σημαντικά τη στρέβλωση της δειγματοληψίας, την υπερχείλιση (overflow) του software, και τη χρονική αβεβαιότητα μεταξύ των καναλιών. Πρόκειται για μια ιδιαίτερα αποδοτική λειτουργία για υψηλό ρυθμό συλλογής με μικρή καθυστέρηση μεταξύ σημάτων. Στην υλοποίηση της παρούσας εργασίας, τα conversion results των ADC1 και ADC2 μεταφέρονται μέσω DMA στη μνήμη, επιτρέποντας συνεχή συλλογή χωρίς μεσολάβηση CPU. Η συγχρονισμένη λειτουργία των δύο ADC παρέχει επίσης πιο αξιόπιστη συμπεριφορά σε σχέση με ακολουθιακές μετατροπές ενεργοποιημένες από software.

3.2.4 Hardware triggering

Η σταθερότητα του ρυθμού δειγματοληψίας αποτελεί άλλον ένα παράγοντα που πρέπει να λαμβάνεται σοβαρά υπόψη σε συστήματα συλλογής υψηλής ταχύτητας. Για τον λόγο αυτό η ενεργοποίηση των ADC δεν πραγματοποιείται μέσω polling software ή διακοπές, αλλά μέσω χρονιστών hardware. Οι ADC ενεργοποιούνται από hardware trigger events τα οποία παράγονται από timer peripherals του μικροελεγκτή. Η χρήση hardware triggering εξασφαλίζει ότι οι μετατροπές ξεκινούν με ακριβή χρονισμό, το sample interval παραμένει σταθερό, και η αστάθεια του χρονισμού της δειγματοληψίας μειώνεται σημαντικά. Σε software-triggered συστήματα, η χρονική στιγμή έναρξης μετατροπής εξαρτάται από τον φόρτο του CPU, τις διακοπές, και την καθυστέρηση του firmware. Το γεγονός αυτό μπορεί να δημιουργήσει μη σταθερά διαλείμματα δειγματοληψίας, κάτι ιδιαίτερα ανεπιθύμητο σε εφαρμογές ψηφιακής επεξεργασίας σημάτων. Η χρήση hardware timers επιτρέπει τη ντετερμινιστική

λειτουργία και σταθερό χρονισμό ανεξάρτητα από τη δραστηριότητα του επεξεργαστή ή του υποσυστήματος του USB.

3.2.5 Χρόνος δειγματοληψίας και χρόνος μετατροπής

Οι ADC του STM32H7 επιτρέπουν ρυθμιζόμενους χρόνους δειγματοληψίας για κάθε κανάλι. Ο χρόνος δειγματοληψίας καθορίζει το χρονικό διάστημα κατά το οποίο το κύκλωμα δειγματοληψίας και συγκράτησης (sample-and-hold) συνδέεται με την είσοδο ώστε να φορτιστεί ο εσωτερικός πυκνωτής. Η επιλογή του ρυθμιζόμενοι χρόνου δειγματοληψίας αποτελεί σημαντικό σχεδιαστικό συμβιβασμό. Μικρός χρόνος δειγματοληψίας επιτρέπει υψηλότερους ρυθμούς δειγματοληψίας, αλλά μπορεί να μειώσει την ακρίβεια όταν η πηγή του σήματος παρουσιάζει υψηλή αντίσταση εξόδου. Αντίθετα, μεγαλύτερος sample time αυξάνει την ακρίβεια της μέτρησης αλλά μειώνει το μέγιστο δυνατό ρυθμό.

Ο συνολικός χρόνος μετατροπής περιλαμβάνει:

- τον χρόνο δειγματοληψίας και συγκράτησης,
- και τον χρόνο εκτέλεσης της διαδικασίας μετατροπής SAR.

Στην παρούσα εργασία οι ρυθμίσεις των ADC επιλέχθηκαν με προτεραιότητα την επίτευξη υψηλής διεκπεραιωτικότητας και σταθερής λειτουργίας.

3.2.6 ADC clocking

Η λειτουργία των ADC εξαρτάται άμεσα από το υποσύστημα ρολογιού μικροελεγκτή και η συχνότητά της επηρεάζει τον ρυθμό δειγματοληψίας , τον χρόνο μετατροπής και την ποιότητα των μετρήσεων.

Στον STM32H7 οι ADC μπορούν να χρονίζονται είτε από ασύγχρονο ρολόι είτε από ρολόι μέσω PLL.

Η επιλογή του ασύγχρονου ρολογιού ADC μειώνει την εξάρτηση από το ρολόι CPU και επιτρέπει πιο σταθερή λειτουργία του υποσυστήματος συλλογής. Ωστόσο, υψηλότερες συχνότητες λειτουργίας αυξάνουν τον ηλεκτρικό θόρυβο, την κατανάλωση ισχύος, και πιθανές μη ιδανικότητες του ADC και, για τον λόγο αυτό, απαιτείται προσεκτικός συμβιβασμός μεταξύ ταχύτητας και ποιότητας μετατροπής.

3.2.7 DMA integration

Η συνεχής λειτουργία των ADC με υψηλό ρυθμό απόκτησης των δειγμάτων δημιουργεί τεράστιο όγκο δεδομένων. Αν ο επεξεργαστής έπρεπε να διαβάζει κάθε αποτέλεσμα μετατροπής μέσω διακοπών ή polling, το φορτίο CPU θα ήταν εξαιρετικά υψηλό· γι’ αυτό χρησιμοποιούνται ελεγκτές DMA ώστε τα αποτελέσματα της μετατροπής να μεταφέρονται απευθείας στη RAM χωρίς συνεχή εμπλοκή του CPU. Η συνεργασία ADC και DMA αποτελεί βασικό στοιχείο της αρχιτεκτονικής του συστήματος. Μετά από κάθε μετατροπή:

1. ο ADC γράφει το αποτέλεσμα στο data register,
2. το DMA ανιχνεύει το συμβάν,
3. και μεταφέρει αυτόματα την τιμή στη RAM.

Η διαδικασία αυτή πραγματοποιείται πλήρως σε επίπεδο hardware και επιτρέπει:

- πολύ υψηλούς ρυθμούς δειγματοληψίας,
- αποφυγή υπερχείλισης CPU,
- και αξιόπιστη λειτουργία.

Στην παρούσα εργασία χρησιμοποιείταιλειτουργία διπλού κυκλικού buffer ώστε η διαδικασία της συλλογής να είναι αδιάλειπτη.

3.2.8 Θόρυβος και ποιότητα μετρήσεων

Η ποιότητα των ADC μετρήσεων επηρεάζεται από πολλούς παράγοντες. Εκτός από τα θεωρητικά χαρακτηριστικά του ADC, σημαντικό ρόλο παίζουν η ποιότητα της τροφοδοσίας, ο θόρυβος του PCB, η γείωση, και η ηλεκτρομαγνητική παρεμβολή. Σε συστήματα υψηλής ταχύτητας εμφανίζονται επίσης θόρυβοι που σχετίζονται με την εναλλαγή ρολογιών, τη δραστηριότητα των μετατροπέων DMA, και την επικοινωνία μέσω USB. Η ταυτόχρονη λειτουργία πολλών high-speed περιφερειακών μπορεί να επηρεάσει τη στάθμη θορύβου των ADC και να μειώσει το ενεργού αριθμού bits. Για τον λόγο αυτό απαιτείται οπωσδήποτε σωστό PCB layout, αποσύζευξη τροφοδοσίας, και διαχωρισμός αναλογικών και ψηφιακών επιστροφών ρεύματος.

3.2.9 Περιορισμοί και σχεδιαστικοί συμβιβασμοί

Παρότι οι ενσωματωμένοι ADC του STM32H7 παρέχουν ιδιαίτερα υψηλές δυνατότητες για ενσωματωμένες εφαρμογές, παρουσιάζουν και ορισμένους περιορισμούς, οι σημαντικότεροι από τους οποίους σχετίζονται με το πραγματικό ENOB σε υψηλούς ρυθμούς δειγματοληψίας, τη λειτουργία της cache, το διαθέσιμο εύρος ζώνης της μνήμης, και τον περιορισμένο ρυθμό μεταφοράς μέσω USB. Επειδή η επίτευξη μέγιστου ρυθμού συχνά συνοδεύεται από αυξημένο θόρυβο, μεγαλύτερη κατανάλωση ισχύος, και δυσκολότερη διαχείριση δεδομένων, η τελική αρχιτεκτονική του υποσυστήματος ADC σχεδιάστηκε, όσο έγινε δυνατό, ως ένας συμβιβασμός μεταξύ ταχύτητας, σταθερότητας και αξιοπιστίας στη μεταφορά δεδομένων.

3.3 Υποσύστημα DMA

Το υποσύστημα DMA (Direct Memory Access) αποτελεί ένα από τα σημαντικότερα στοιχεία της αρχιτεκτονικής του συστήματος συλλογής δεδομένων που αναπτύχθηκε στην παρούσα εργασία. Η χρήση DMA είναι απαραίτητη σε εφαρμογές όπως αυτή της παρούσας εργασίας, καθώς επιτρέπει τη μεταφορά μεγάλου όγκου δεδομένων μεταξύ περιφερειακών και μνήμης χωρίς συνεχή εμπλοκή του επεξεργαστή. Στην παρούσα υλοποίηση οι ADC λειτουργούν με ρυθμούς δειγματοληψίας που μπορούν να φτάσουν το 1 MSPS ανά κανάλι. Ο συνολικός όγκος δεδομένων που παράγεται είναι ιδιαίτερα μεγάλος και η εξυπηρέτηση των μεταφορών αποκλειστικά μέσω CPU θα οδηγούσε σε υπερβολική επιβάρυνση ως προς την επεξεργασία, υψηλή συχνότητα διακοπών και μείωση του πραγματικού χρόνου λειτουργίας.

Για τον λόγο αυτό χρησιμοποιούνται πολλαπλοί ελεγκτές DMA του μικροελεγκτή STM32H743ZI ώστε η μεταφορά των αποτελεσμάτων μετατροπής προς τη RAM να πραγματοποιείται πλήρως σε επίπεδο hardware. Η χρήση DMA επιτρέπει συνεχή acquisition, ελαχιστοποίηση φορτίου CPU, ντετερμινιστική λειτουργία, και υψηλή διεκπεραιωτικότητα, καθώς συνεργάζεται στενά με τους ADC, τη RAM, το υποσύστημα USB, και το υποσύστημα διακοπής του Cortex-M7. Για όλους αυτούς τους λόγους η σωστή διαχείριση των μεταφορών DMA αποτελεί κρίσιμο στοιχείο της συνολικής αρχιτεκτονικής του συστήματος.

3.3.1 Θεωρία λειτουργίας DMA

Σε ένα τυπικό ενσωματωμένο σύστημα, η μεταφορά δεδομένων μεταξύ περιφερειακών και μνήμης πραγματοποιείται μέσω του επεξεργαστή. Ο CPU διαβάζει δεδομένα από έναν καταχωρητή

περιφερειακού και στη συνέχεια τα γράφει στη RAM. Η διαδικασία αυτή φαίνεται απλή αλλά είναι ιδιαίτερα αναποτελεσματική όταν οι μεταφορές δεδομένων είναι συνεχείς και μεγάλης ταχύτητας. Ο ελεγκτής DMA λειτουργεί ως ένας ανεξάρτητος μηχανισμός μεταφοράς δεδομένων που μπορεί να προσπελαύνει καταχωρητές περιφερειακών, SRAM, και δίαυλους μνήμης, χωρίς να απαιτείται συνεχής παρέμβαση του CPU. Αυτό που συμβαίνει στην πράξη είναι ότι ο επεξεργαστής ρυθμίζει αρχικά τη διεύθυνση πηγής, τη διεύθυνση προορισμού, το μέγεθος μεταφοράς, και τον τρόπο λειτουργίας του DMA. Μετά την ενεργοποίηση, ο ελεγκτής DMA αναλαμβάνει αυτόνομα τη μεταφορά των δεδομένων και μόνο όταν ολοκληρωθεί η διαδικασία ή υπάρξει κάποιο συμβάν, δημιουργείται διακοπή προς τον επεξεργαστή. Η αρχιτεκτονική αυτή μειώνει δραστικά τον αριθμό διακοπών, τη χρήση CPU, και τη χρονική ασυνέπεια του firmware. Σε εφαρμογές υψηλής ταχύτητας το DMA είναι ουσιαστικά απαραίτητο, καθώς χωρίς αυτό ο επεξεργαστής δεν θα μπορούσε να εξυπηρετήσει τον απαιτούμενο ρυθμό δεδομένων.

3.3.2 Αρχιτεκτονική DMA του STM32H743ZI

Ο STM32H743ZI διαθέτει πολλαπλά υποσυστήματα DMA σχεδιασμένα για εφαρμογές υψηλής απόδοσης. Η αρχιτεκτονική DMA του STM32H7 είναι σημαντικά πιο σύνθετη σε σχέση με παλαιότερες οικογένειες STM32 λόγω των πολλαπλών τομέων μνήμης, της cache και του υψηλού εύρους ζώνης της μνήμης του Cortex-M7. Ο μικροελεγκτής διαθέτει DMA1, DMA2, BDMA και MDMA. Τα DMA1 και DMA2 χρησιμοποιούνται κυρίως για περιφερειακά υψηλής ταχύτητας, ενώ το BDMA είναι ένας απλούστερος ελεγκτής χαμηλότερου εύρους ζώνης. Το MDMA χρησιμοποιείται κυρίως για μεταφορές ‘μνήμη-σε-μνήμη’ και προηγμένο χειρισμό δεδομένων.

Στην παρούσα εργασία χρησιμοποιούνται DMA1, DMA2, και BDMA,για την εξυπηρέτηση των τριών ADC. Οι ADC1 και ADC2 συνδέονται με DMA υψηλότερης διεκπεραιωτικότητας, ενώ ο ADC3 εξυπηρετείται από το BDMA. Η κατανομή αυτή επιλέχθηκε ώστε να αποφεύγονται οι συγκρούσεις πόρων, να μειώνεται η *σύγκρουση διαύλου* (bus contention), και να επιτυγχάνεται σταθερή λειτουργία του pipeline της διαδικασίας συλλογής.

3.3.3 DMA και ADC integration

Η συνεργασία μεταξύ ADC και DMA αποτελεί βασικό στοιχείο της αρχιτεκτονικής του συστήματος. Μετά την ολοκλήρωση κάθε μετατροπής ADC, δημιουργείται ένα συμβάν αιτήματος DMA, το οποίο ενεργοποιεί αυτόματα τη μεταφορά του αποτελέσματος της μετατροπής προς τη RAM. Η διαδικασία αυτή πραγματοποιείται πλήρως σε επίπεδο hardware χωρίς να απαιτείται μεσολάβηση software. Ο μηχανισμός λειτουργίας είναι ο εξής:

1. Ο ADC ολοκληρώνει τη μετατροπή.
2. Το αποτέλεσμα μετατροπής αποθηκεύεται στο καταχωρητή δεδομένων ADC.
3. Δημιουργείται αίτημα DMA.
4. Ο ελεγκτής DMA διαβάζει την τιμή.
5. Η τιμή γράφεται στη RAM.

Η διαδικασία επαναλαμβάνεται συνεχώς για κάθε νέο δείγμα.

Η αρχιτεκτονική αυτή επιτρέπει στους ADC να λειτουργούν συνεχώς με υψηλό ρυθμό δειγματοληψίας χωρίς ο επεξεργαστής να χρειάζεται να εξυπηρετεί κάθε μετατροπή ξεχωριστά. Ενώ αν η συλλογή δεδομένων πραγματοποιούνταν μέσω διακοπών, τότε κάθε δείγμα θα δημιουργούσε διακοπή, με

συνέπεια ο CPU να εξυπηρετεί εκατομμύρια διακοπές ανά δευτερόλεπτο και, τελικά, η συνολική απόδοση του συστήματος θα ήταν σε μη αποδεκτά επίπεδα. Το DMA επιλύει αυτό το πρόβλημα πραγματοποιώντας μεταφορές δεδομένων σε μπλοκ.

3.3.4 Circular mode

Για εφαρμογές συνεχούς συλλογής δεδομένων είναι απαραίτητη η δυνατότητα συνεχούς επαναχρησιμοποίησης των buffers χωρίς διακοπή της δειγματοληψίας. Για τον λόγο αυτό χρησιμοποιείται η κυκλική λειτουργία (circular mode) DMA. Σε αυτή τη λειτουργία, όταν ο DMA φτάσει στο τέλος του buffer **δεν** σταματά, αλλά επιστρέφει αυτόματα στην αρχή του buffer, συνεχίζοντας τη μεταφορά δεδομένων. Η τεχνική αυτή επιτρέπει όχι μόνο συνεχή λειτουργία και σταθερό acquisition pipeline, αλλά και ελαχιστοποίηση της επιβάρυνσης του software, γιατί ο επεξεργαστής δεν χρειάζεται να επανεκκινεί συνεχώς τις μεταφορές DMA, γεγονός που βελτιώνει σημαντικά τη συμπεριφορά του συστήματος. Το circular mode – που είναι ιδιαίτερα χρήσιμο σε εφαρμογές συνεχούς ροής, επεξεργασίας σε πραγματικό χρόνο και συνεχούς παρακολούθησης, στην παρούσα εργασία χρησιμοποιείται σε συνδυασμό με το διπλό buffering (βλ. παρακάτω) ώστε να επιτρέπεται ταυτόχρονη συλλογή και επεξεργασία δεδομένων.

3.3.5 Διπλό buffering

Το λεγόμενο διπλό buffering χρησιμοποιείται ευρέως σε συστήματα υψηλού ρυθμού απόδοσης ώστε να αποφεύγεται η απώλεια δεδομένων κατά τη διάρκεια επεξεργασίας ή μεταφοράς. Πρακτικά σημαίνει ότι ο buffer χωρίζεται λογικά σε δύο τμήματα: το πρώτο μισό, και το δεύτερο μισό, και ενώ ο DMA γράφει δεδομένα στο ένα τμήμα, το firmware μπορεί ταυτόχρονα να επεξεργάζεται, να πακετάρει, ή να μεταδίδει τα δεδομένα του άλλου τμήματος. Όταν γεμίσει το πρώτο μισό του buffer δημιουργείται διακοπή ‘μισή μεταφορά’. Όταν γεμίσει ολόκληρος ο buffer δημιουργείται διακοπή ‘ολοκλήρωση μεταφοράς’.

Η τεχνική αυτή επιτρέπει αδιάλειπτη συλλογή δεδομένων με παράλληλη επεξεργασία τους, και αποδοτική χρήση της RAM, ενώ χωρίς αυτή ο επεξεργαστής θα έπρεπε να σταματά προσωρινά τη συλλογή κατά τη διάρκεια επεξεργασίας των δεδομένων, κάτι που θα οδηγούσε σε απώλεια δεδομένων.

3.3.6 Interrups μισής και ολοκληρωμένης μεταφοράς

Παρότι το DMA μειώνει δραστικά τις διακοπές, εξακολουθούν να χρησιμοποιούνται συγκεκριμένα συμβάντα DMA ώστε το firmware να γνωρίζει πότε υπάρχουν νέα δεδομένα διαθέσιμα. Στην παρούσα εργασία χρησιμοποιούνται Διακοπές ‘μισή μεταφορά’ και και Διακοπές ‘ολοκλήρωση μεταφοράς’. Η πρώτη ενεργοποιείται όταν γεμίσει το πρώτο μισό του buffer και η δεύτερη όταν ολοκληρωθεί η πλήρωση ολόκληρου του buffer. Με αυτό τον τρόπο το firmware μπορεί να επεξεργάζεται δεδομένα σε μπλοκ και όχι μόνο μεμονωμένα δείγματα. Η επεξεργασία με βάση μπλοκ δεδομένων είναι πολύ πιο αποδοτική διότι μειώνει την επιβάρυνση που προκαλούν οι διακοπές, αυξάνει την αποδοτικότητα της cache, και βελτιώνει την πακετοποίηση του USB. Η χρήση μεγαλύτερων data blocks μειώνει επίσης τη σχετική επιβάρυνση από function calls, μηχανισμούς συγχρονισμού, και ‘συναλλαγές’ του USB.

3.3.7 Εύρος ζώνης μνήμης και διαιτησία διαύλου

Σε συστήματα υψηλής διεκπεραιωτικότητας η πρόσβαση στη μνήμη αποτελεί κρίσιμο παράγοντα απόδοσης. Στην παρούσα εργασία πολλαπλά υποσυστήματα προσπελαύνουν ταυτόχρονα τη RAM:

- οι ADC μέσω DMA,

- ο CPU,
- το περιφερειακό USB,
- και ενδεχομένως η cache.

Η ταυτόχρονη πρόσβαση δημιουργεί διαμάχη διαύλου (bus contention) η οποία απαιτεί διαιτησία (arbitration) μεταξύ των διαφορετικών masters του συστήματος. Ο STM32H7 διαθέτει ιδιαίτερα σύνθετη αρχιτεκτονική μνήμης με δίαυλο AXI, πολλαπλούς τομείς SRAM, και διαφορετικές περιοχές μνήμης με διαφορετικά χαρακτηριστικά πρόσβασης. Η σωστή επιλογή περιοχής μνήμης για τα DMA buffers επηρεάζει σημαντικά την αποδοτικότητα, την ταχύτητα και τη σταθερότητα του συνολικού pipeline. Σε ορισμένες περιοχές μνήμης η πρόσβαση DMA δεν υποστηρίζεται πλήρως ή παρουσιάζει μειωμένη απόδοση λόγω διεπίδρασης της cache.

3.3.8 Προβλήματα συνοχής της Cache

Η χρήση cache στον πυρήνα ARM Cortex-M7 αυξάνει σημαντικά την απόδοση του επεξεργαστή, αλλά δημιουργεί προβλήματα συνοχής όταν χρησιμοποιούνται μεταφορές DMA καθώς Ο DMA προσπελαύνει απευθείας τη φυσική RAM χωρίς να ενημερώνει την cache του CPU, με αποτέλεσμα να μπορεί να συμβεί το εξής: ο DMA να έχει γράψει νέα δεδομένα στη RAM αλλά ο CPU να συνεχίζει να βλέπει παλιά cached δεδομένα. Το πρόβλημα αυτό είναι ιδιαίτερα σημαντικό σε acquisition εφαρμογές όπου η αξιοπιστία των δεδομένων είναι κρίσιμη. Για την αποφυγή cache coherency προβλημάτων μπορούν να χρησιμοποιηθούν διάφορες τεχνικές:

- Ακύρωση cache μετά από DMA writes,
- χρήση περιοχών της μνήμης non-cacheable,
- ή προσεκτική ‘ευθυγράμμιση’ της μνήμης.

Στην παρούσα εργασία απαιτήθηκε ιδιαίτερη προσοχή στη διαχείριση της cache ώστε να διασφαλιστεί η ορθή λειτουργία του συνολικού pipeline.

3.3.9 DMA και USB streaming

Όπως ήδη αναφέραμε, το υποσύστημα DMA συνεργάζεται στενά με το υποσύστημα USB. Τα δεδομένα που παράγονται από τους ADC μεταφέρονται αρχικά στη RAM μέσω DMA και στη συνέχεια αποστέλλονται στον υπολογιστή μέσω USB. Το pipeline λειτουργίας είναι το εξής λοιπόν:

1. Συλλογή μέσω ADC,
2. Μεταφορά DMA στη RAM,
3. πακετοποίηση,
4. Μετάδοση μέσω USB.

Το pipeline πρέπει να λειτουργεί συνεχώς χωρίς σημεία συμφόρησης ώστε να αποφεύγεται υπερχείλιση των buffers. Ωστόσο, το USB Full Speed διαθέτει σημαντικά μικρότερο εύρος ζώνης από τον συνολικό ρυθμό παραγωγής δεδομένων των ADC και γι’ αυτό σε υψηλά sample rates χρησιμοποιείται προσωρινή αποθήκευση στη RAM, ενώ σε χαμηλότερους ρυθμούς υποστηρίζεται real-time streaming. Η διαχείριση των DMA buffers είναι καθοριστική για τη σωστή λειτουργία και των δύο τρόπων λειτουργίας.

3.3.10 Περιορισμοί και σχεδιαστικοί συμβιβασμοί

Παρότι το DMA προσφέρει σημαντικά πλεονεκτήματα, παρουσιάζει και ορισμένους περιορισμούς. Συγκεκριμένα, η συνεχής λειτουργία πολλών ροών DMA δημιουργεί αυξημένη κίνηση μνήμης, διαμάχη διαύλου και μεγαλύτερη κατανάλωση ισχύος. Επιπλέον, η χρήση πολύ μικρών buffer αυξάνει τη συχνότητα διακοπών, ενώ, αντίθετα, η χρήση πολύ μεγάλων buffers αυξάνει τη καθυστέρηση, τη χρήση της RAM, και τον χρόνο επεξεργασίας. Για τον λόγο αυτό η επιλογή μεγέθους buffers αποτελεί σημαντικό σχεδιαστικό συμβιβασμό μεταξύ διακίνησης δεδομένων, καθυστέρησης, και απασχόλησης της μνήμης. Η τελική αρχιτεκτονική του υποσυστήματος DMA σχεδιάστηκε για την παρούσα εργασία έτσι ώστε να επιτυγχάνει σταθερή λειτουργία και αξιόπιστη συλλογή δεδομένων ακόμη και σε υψηλούς ρυθμούς acquisition.

Κεφάλαιο 4ο: Υλοποίηση Firmware

4.1 Δομή και Οργάνωση του Firmware

Το firmware της συσκευής που υλοποιήθηκε στο πλαίσιο της παρούσας εργασίας αναπτύχθηκε σε γλώσσα C, χρησιμοποιώντας το περιβάλλον ανάπτυξης STM32CubeIDE, και βασίζεται στον μικροελεγκτή STM32H743ZI. Οι κύριοι σκοποί της αρχιτεκτονικής του λογισμικού μπορούν να συνοψιστούν στα εξής: (α) διαχωρισμός των λειτουργιών χαμηλού επιπέδου από τις λειτουργίες συλλογής και μεταφοράς δεδομένων, (β) ευκολότερη ανάπτυξη, συντήρηση και επέκταση του κώδικα. Όσον αφορά τις κύριες βιβλιοθήκες λογισμικού που χρησιμοποιήθηκαν αυτές ήταν τρεις, και συγκεκριμένα:

Η βιβλιοθήκη CMSIS, η οποία παρέχει τις βασικές δηλώσεις και τους ορισμούς του πυρήνα ARM Cortex-M7. Η CMSIS δεν χρησιμοποιήθηκε άμεσα από τον κώδικα της εφαρμογής, αλλά αποτελεί απαραίτητο τμήμα του λογισμικού που παράγεται από το STM32CubeMX και χρησιμοποιείται από τα χαμηλότερα επίπεδα του HAL και του startup code.

Η δεύτερη βιβλιοθήκη είναι η STM32 HAL Driver Library, η οποία παρέχει αφαιρετικό επίπεδο πρόσβασης στα περιφερειακά του μικροελεγκτή. Μέσω αυτής της βιβλιοθήκης υλοποιείται η αρχικοποίηση και η διαχείριση των ADC, DMA, USB και των υπολοίπων περιφερειακών.

Τέλος, η τρίτη βιβλιοθήκη είναι η STM32 USB Device Library, η οποία χρησιμοποιήθηκε ως βάση για την ανάπτυξη της custom USB class.

Η συνολική δομή του firmware ακολουθεί πολυεπίπεδη αρχιτεκτονική. Τα επίπεδα αυτά επικοινωνούν μεταξύ τους μέσω σαφώς καθορισμένων διεπαφών (interfaces), επιτρέποντας τον διαχωρισμό των λειτουργιών που εξαρτώνται από το hardware από τις λειτουργίες της εφαρμογής.

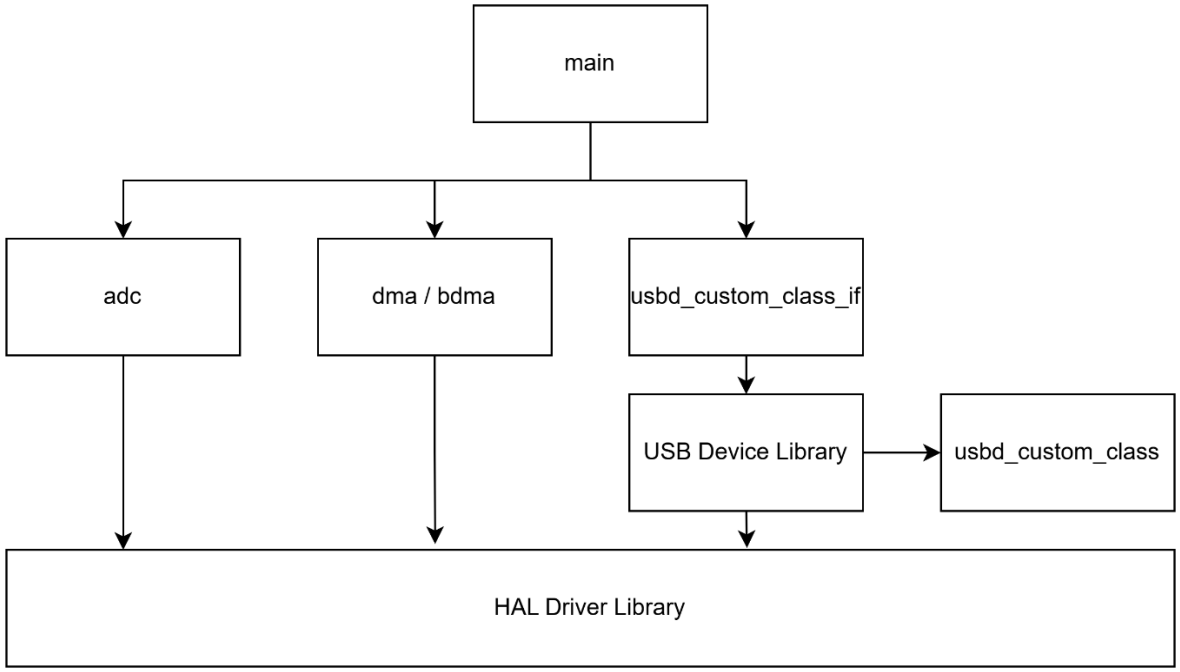
4.1.1 CubeMX Generated Components

Το STM32CubeMX χρησιμοποιήθηκε για τη βασική διαμόρφωση του μικροελεγκτή και την παραγωγή του αρχικού κώδικα. Από το εργαλείο αυτό δημιουργήθηκαν τα modules αρχικοποίησης των περιφερειακών που χρησιμοποιούνται από το σύστημα συλλογής.

Το αρχείο adc.c περιλαμβάνει τη διαμόρφωση των τριών ADC, τις ρυθμίσεις λειτουργίας τους, τους χρονισμούς μετατροπής καθώς και τις απαραίτητες συνδέσεις με τα αντίστοιχα DMA streams.

Το αρχείο dma.c περιλαμβάνει τη διαμόρφωση των DMA controllers που χρησιμοποιούνται από τους ADC1 και ADC2, ενώ το bdma.c περιλαμβάνει τη διαμόρφωση του ελεγκτή BDMA που χρησιμοποιείται από τον ADC3.

Η αυτόματη παραγωγή του αρχικού κώδικα από το CubeMX επέτρεψε την ταχύτερη δημιουργία μιας λειτουργικής βάσης πάνω στην οποία υλοποιήθηκαν οι ειδικές απαιτήσεις του συστήματος συλλογής. Παρόλο που τα συγκεκριμένα modules παράχθηκαν αυτόματα, αποτελούν βασικό τμήμα της αρχιτεκτονικής του firmware καθώς είναι υπεύθυνα για τη σωστή ενεργοποίηση των ADC και DMA πριν ξεκινήσει η διαδικασία δειγματοληψίας.



Σχήμα 4.1: Δομή Firmware

4.1.2 Acquisition Layer

Το επίπεδο συλλογής (acquisition layer) αποτελεί το σημαντικότερο τμήμα του firmware επειδή είναι υπεύθυνο για τη συλλογή των μετρήσεων από τους ADC και την προσωρινή αποθήκευσή τους στη μνήμη. Η δειγματοληψία πραγματοποιείται πλήρως σε επίπεδο hardware μέσω της συνεργασίας ADC και DMA: οι ADC παράγουν συνεχώς νέα δείγματα, τα οποία στη συνέχεια μεταφέρονται αυτόματα προς τη RAM χωρίς καμία παρέμβαση από τον επεξεργαστή. Για τους ADC1 και ADC2 χρησιμοποιούνται ανεξάρτητα DMA streams, ενώ για τον ADC3 χρησιμοποιείται BDMA stream.

Οι buffers των ADC ορίζονται ως εξής:

```
ALIGN_32BYTES(uint16_t adc1_buf[ADC_BUF_SIZE]);
ALIGN_32BYTES(uint16_t adc2_buf[ADC_BUF_SIZE]);
__attribute__((section(".RAM_D3"))) uint16_t adc3_buf[ADC_BUF_SIZE];
```

Η χρήση του ALIGN_32BYTES σχετίζεται με τις απαιτήσεις του cache subsystem του Cortex-M7 και εξασφαλίζει ότι οι buffers ευθυγραμμίζονται σε cache line boundaries.

Ιδιαίτερο ενδιαφέρον παρουσιάζει η τοποθέτηση του buffer του ADC3 στη περιοχή μνήμης RAM_D3. Η επιλογή αυτή δεν έγινε για λόγους οργάνωσης του κώδικα αλλά επειδή η αρχιτεκτονική του DMA του STM32H7 επέφερε περιορισμούς. Τέλος, η διαχείριση των διαφορετικών τομέων μνήμης του STM32H743 αποτέλεσε μία από τις σημαντικότερες προκλήσεις της ανάπτυξης, καθώς διαφορετικοί DMA controllers έχουν πρόσβαση σε διαφορετικές περιοχές μνήμης.

4.1.3 Memory layout και τροποποιήσεις Linker

Κατά την ανάπτυξη του firmware απαιτήθηκαν σημαντικές τροποποιήσεις στα linker scripts.

Ο STM32H743 διαθέτει πολλαπλές περιοχές SRAM κατανεμημένες σε διαφορετικούς τομείς μνήμης:

- DTCM RAM

- AXI SRAM (RAM_D1)
- SRAM1/2/3 (RAM_D2)
- SRAM4 (RAM_D3)

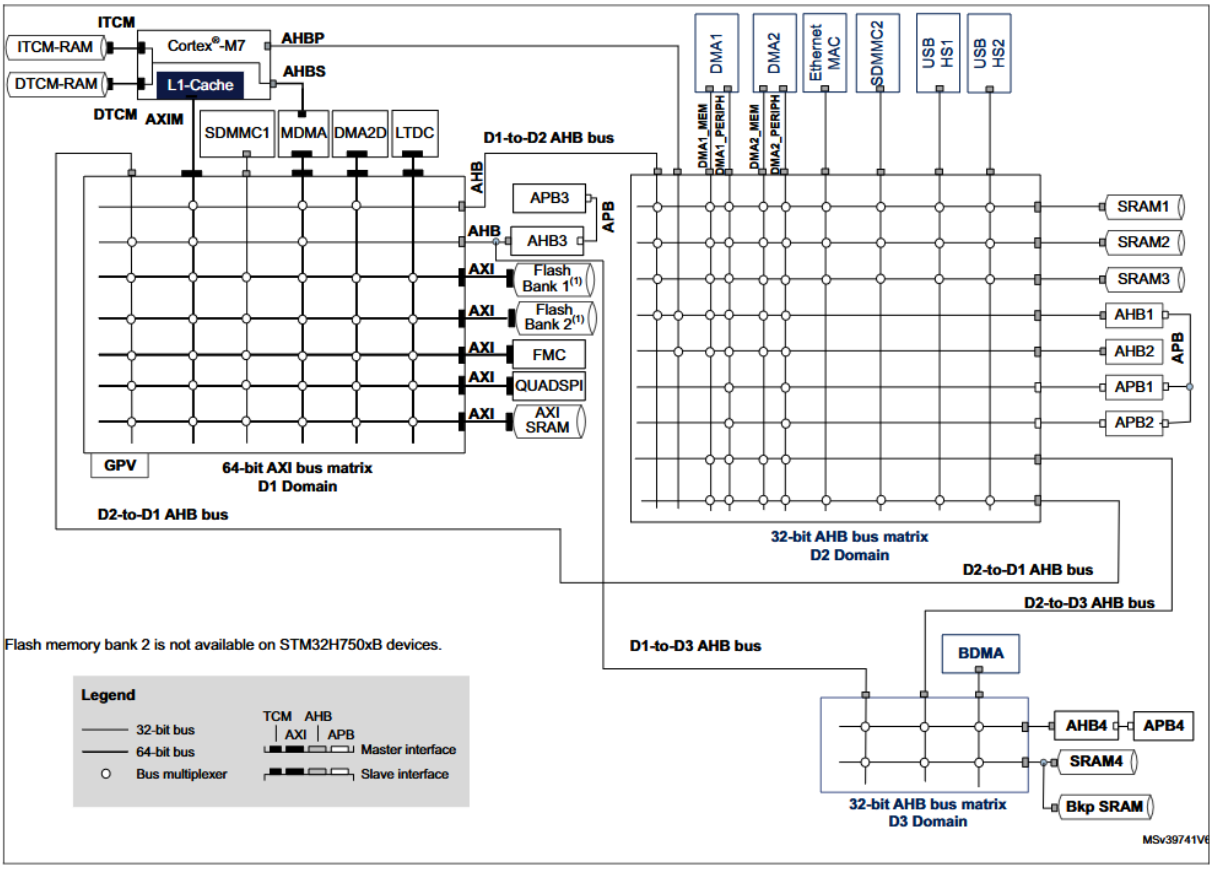
Η αρχική διάταξη όμως που παράγεται από το CubeMX δεν ήταν επαρκής για τις ανάγκες της εφαρμογής και γι' αυτό τροποποιήθηκαν τα αρχεία: STM32H743ZITX_FLASH.ld και STM32H743ZITX_RAM.ld ώστε να οριστεί ξεχωριστή ενότητα (section) για τη μνήμη RAM_D3.

```
  /* Uninitialized data section */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss
section */
    _sbss = .;         /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global symbol at bss end */
    __bss_end__ = _ebss;
  } >RAM_D1

  .RAM_D3 :
  {
    . = ALIGN(4);
    *(.RAM_D3)
    *(.RAM_D3*)
    . = ALIGN(4);
  } >RAM_D3
```

Η τροποποίηση αυτή επέτρεψε τη ρητή τοποθέτηση συγκεκριμένων buffers σε memory region προσβάσιμο από τον ελεγκτή BDMA. Η διαχείριση των τομέων μνήμης αποδείχθηκε κρίσιμη για τη σωστή λειτουργία του συστήματος και αποτέλεσε ένα από τα πιο απαιτητικά στάδια της ανάπτυξης.



Σχήμα 4.2: Memory and bus architecture (STMicroelectronics 2023)

4.1.4 USB Transport Layer

Για την επικοινωνία με τον υπολογιστή αναπτύχθηκε USB ειδικής κλάσης βασισμένης στη βιβλιοθήκη STM USB Device. Η υλοποίηση βρίσκεται κυρίως στο αρχείο: usbd_custom_class.c, το οποίο προστέθηκε εξ ολοκλήρου στη βιβλιοθήκη USB Device.

Η συσκευή δηλώνεται ως vendor specific USB interface:

```
bInterfaceClass = 0xFF
```

και χρησιμοποιεί δύο τελικά σημεία μαζικής επεξεργασίας:

- ένα Bulk IN endpoint για μεταφορά δεδομένων προς τον υπολογιστή
- ένα Bulk OUT endpoint για λήψη δεδομένων από τον υπολογιστή

Η χρήση των τελικών σημείων μαζικής επεξεργασίας επιλέχθηκε λόγω: μεγαλύτερης διεκπεραιωτικότητας, μικρότερης επιβάρυνσης του πρωτοκόλλου, και αξιόπιστης μεταφοράς δεδομένων.

Κατά την αρχικοποίηση ανοίγονται και τα δύο τελικά σημεία και προετοιμάζεται ο receive buffer της συσκευής.

Ο ρόλος του module ήταν να διαχειρίζεται τη μετάδοση και υποδοχή των πακέτων, την κατάσταση των τελικών σημείων και τα συμβάντα ολοκλήρωσης μετάδοσης.

4.1.5 USB Interface Layer

Πάνω από το USB class layer αναπτύχθηκε ένα δεύτερο επίπεδο αφαίρεσης (abstraction) στο αρχείο: usbd_custom_class_if.c. Το module αυτό λειτουργεί ως γέφυρα μεταξύ της USB βιβλιοθήκης και της εφαρμογής acquisition.

Η βασική συνάρτηση μετάδοσης είναι:

```
custom_class_transmit_data(...)
```

η οποία χρησιμοποιείται από το application layer για την αποστολή δεδομένων προς τον υπολογιστή.

Παράλληλα υλοποιούνται callbacks για:

- ολοκλήρωση μετάδοσης (transmit complete)
- υποδοχή πακέτου (packet reception)
- αρχικοποίηση διεπαφής (interface initialization)
- αποαρχικοποίηση διεπαφής (interface deinitialization)

Ιδιαίτερα σημαντικός είναι η callback για την ολοκλήρωση μετάδοσης, καθώς μέσω αυτού μηδενίζεται η μεταβλητή usb_busy και επιτρέπεται η μετάδοση του επόμενου packet.

4.1.6 Main Application Layer

Το ανώτερο επίπεδο του firmware βρίσκεται στο main loop. Η εφαρμογή λειτουργεί ως state machine που παρακολουθεί δύο flags: half_ready και full_ready, τα οποία ενεργοποιούνται από τα DMA interrupts. Όταν ολοκληρωθεί η πλήρωση του πρώτου μισού του buffer ενεργοποιείται το half_ready. Όταν ολοκληρωθεί η πλήρωση του δεύτερου μισού ενεργοποιείται το full_ready. Ο κύριος βρόχος (main loop) ελέγχει επίσης τη μεταβλητή usb_busy ώστε να αποφεύγεται η εκκίνηση νέας USB μεταφοράς πριν ολοκληρωθεί η προηγούμενη. Τα δεδομένα των τριών ADC αντιγράφονται σε δομές πακέτων USB και αποστέλλονται στον υπολογιστή. Ουσιαστικά η λειτουργία είναι:

ADC → DMA → RAM → USB Packet Buffer → USB Endpoint → PC

4.2 Ροή δεδομένων του Firmware

Η ροή δεδομένων στο firmware αποτελεί το κεντρικό σημείο λειτουργίας του συστήματος συλλογής. Σκοπός της είναι η μεταφορά των αναλογικών μετρήσεων από τους ADC προς τον υπολογιστή με όσο το δυνατόν μικρότερη συμμετοχή του επεξεργαστή και χωρίς απώλεια δειγμάτων. Η συνολική διαδρομή που ακολουθούν τα δεδομένα μπορεί να περιγραφεί ως εξής:

ADC conversion → DMA transfer → double buffer → half/full ready flag → USB packetization → USB bulk transmission → host PC

Στην παρούσα υλοποίηση δεν χρησιμοποιείται ξεχωριστή ουρά δεδομένων. Αντί για queue, χρησιμοποιείται μηχανισμός **double buffering**, όπου κάθε ADC διαθέτει buffer χωρισμένο λογικά σε δύο τμήματα. Όταν το DMA ολοκληρώσει τη μεταφορά του πρώτου μισού, ενεργοποιείται το αντίστοιχο flag `half_ready`. Όταν ολοκληρωθεί η πλήρωση ολόκληρου του buffer, ενεργοποιείται το `full_ready`. Με αυτόν τον τρόπο το firmware γνωρίζει ποιο μισό του buffer περιέχει έγκυρα δεδομένα προς αποστολή, ενώ το DMA μπορεί να συνεχίζει τη συλλογή νέων δειγμάτων στο άλλο τμήμα.

4.2.1 Γενική περιγραφή της ροής δεδομένων

Η λειτουργία του firmware έχει σχεδιαστεί ώστε η συλλογή των δεδομένων να πραγματοποιείται κυρίως σε επίπεδο hardware. Οι ADC εκτελούν τις μετατροπές, οι ελεγκτές DMA μεταφέρουν αυτόματα τα αποτελέσματα στη μνήμη και ο επεξεργαστής παρεμβαίνει μόνο όταν υπάρχει έτοιμο μπλοκ δεδομένων προς επεξεργασία ή αποστολή.
Η προσέγγιση αυτή είναι απαραίτητη λόγω του υψηλού ρυθμού δειγματοληψίας. Σε ρυθμούς της τάξης του 1 MSPS ανά κανάλι, ο επεξεργαστής δεν μπορεί να διαχειρίζεται κάθε δείγμα ξεχωριστά. Αντίθετα, πρέπει να εργάζεται σε μπλοκ δεδομένων, ώστε να μειώνεται ο αριθμός των διακοπών (interrupts) και να μειώνεται ο φόρτος του προγράμματος.
Για τον λόγο αυτό, η εφαρμογή δεν αντιμετωπίζει κάθε δείγμα ως ανεξάρτητο γεγονός. Τα δείγματα συγκεντρώνονται πρώτα στους buffers DMA και στη συνέχεια αντιγράφονται σε transmission buffers USB με διαδοχική μορφή τριών καναλιών. Έτσι, κάθε πακέτο USB περιέχει συγχρονισμένες τιμές των τριών ADC.

4.2.2 Μετατροπή ADC

Το πρώτο στάδιο της ροής δεδομένων είναι η μετατροπή των αναλογικών σημάτων σε ψηφιακές τιμές. Οι τρεις ADC του μικροελεγκτή λειτουργούν παράλληλα, με τους ADC1 και ADC2 να είναι συγχρονισμένοι μέσω hardware dual mode, ενώ ο ADC3 λειτουργεί ως ανεξάρτητος μετατροπέας. Η εκκίνηση των μετατροπών γίνεται με σταθερό χρονισμό, ώστε τα δείγματα να λαμβάνονται σε προκαθορισμένα χρονικά διαστήματα. Η χρήση hardware triggering είναι σημαντική επειδή αποφεύγει τις καθυστερήσεις που θα εισάγονταν αν η εκκίνηση των μετατροπών γινόταν μέσω software. Με αυτόν τον τρόπο το υποσύστημα συλλογής παραμένει ανεξάρτητο από τον φόρτο του CPU, τα συμβάντα του USB και τις υπόλοιπες διακοπές.
Κάθε φορά που ένας ADC ολοκληρώνει μια μετατροπή, το αποτέλεσμα αποθηκεύεται στο αντίστοιχο καταχωρητή δεδομένων (data register) του περιφερειακού. Από εκεί δεν διαβάζεται απευθείας από τον επεξεργαστή, αλλά παραλαμβάνεται από τον αντίστοιχο ελεγκτή DMA.

4.2.3 Μεταφορά DMA

Το δεύτερο στάδιο είναι η αυτόματη μεταφορά των ADC αποτελεσμάτων στη RAM μέσω DMA. Κάθε ADC συνδέεται με αντίστοιχο DMA μηχανισμό. Οι ADC1 και ADC2 εξυπηρετούνται από DMA streams, ενώ ο ADC3 χρησιμοποιεί BDMA. Η χρήση του BDMA για τον ADC3 επηρέασε και τη χωροθέτηση του αντίστοιχου buffer στη μνήμη, καθώς ο buffer του ADC3 τοποθετήθηκε ειδικά στη RAM_D3 περιοχή. Το σημαντικό σημείο εδώ είναι ότι ο CPU δεν συμμετέχει στη μεταφορά κάθε sample. Ο ρόλος του περιορίζεται στη ρύθμιση του DMA και στη διαχείριση των έτοιμων blocks όταν αυτά ολοκληρωθούν.

4.2.4 Double buffer αντί για queue

Στην παρούσα υλοποίηση δεν χρησιμοποιείται κλασική ουρά δεδομένων. Δεν υπάρχει δηλαδή ένας δυναμικός μηχανισμός enqueue/dequeue όπου τα μπλοκ εισάγονται και εξάγονται από queue. Αντίθετα, χρησιμοποιείται στατικός μηχανισμός double buffering. Κάθε ADC buffer χωρίζεται λογικά σε δύο μισά. Το πρώτο μισό περιέχει τα δείγματα από τη θέση `0` έως `HALF_SAMPLES - 1`, ενώ το δεύτερο μισό περιέχει τα δείγματα από τη θέση `HALF_SAMPLES` έως `ADC_BUF_SIZE - 1`. Όταν το DMA ολοκληρώσει την πλήρωση του πρώτου μισού, το firmware θέτει το flag `half_ready`. Όταν ολοκληρωθεί και η πλήρωση του δεύτερου μισού, τίθεται το flag `full_ready`. Η τεχνική αυτή λειτουργεί σαν ένας απλός μηχανισμός παραγωγού-καταναλωτή. Ο παραγωγός είναι το DMA, το οποίο γεμίζει συνεχώς τους buffers. Ο καταναλωτής είναι το main loop, το οποίο ελέγχει τα flags και αποστέλλει το αντίστοιχο μισό buffer μέσω USB. Σε αντίθεση με μια queue, η λύση αυτή είναι απλούστερη και πιο deterministic. Δεν απαιτεί δυναμική διαχείριση μνήμης, δεν έχει κόστος εισαγωγής και εξαγωγής στοιχείων και είναι κατάλληλη για σταθερό ρυθμό παραγωγής δεδομένων.

4.2.5 Half-transfer interrupt

Όταν το DMA ολοκληρώσει τη μεταφορά του πρώτου μισού του buffer, δημιουργείται η αντίστοιχη διακοπή half-transfer interrupt, η οποία δεν χρησιμοποιείται για βαριά επεξεργασία δεδομένων. Ο βασικός της ρόλος είναι να ενημερώσει την εφαρμογή ότι το πρώτο μισό των ADC buffers περιέχει πλέον έγκυρα δεδομένα. Στο σημείο αυτό το firmware θα θέσει το flag `half_ready`. Η πραγματική επεξεργασία και αποστολή των δεδομένων δεν γίνεται μέσα στον χειριστή διακοπών (interrupt handler), αλλά στον κύριο βρόχο (main loop). Αυτή η επιλογή είναι σημαντική, επειδή μειώνει τη διάρκεια εκτέλεσης των interrupt routines και διατηρεί το σύστημα πιο σταθερό. Με άλλα λόγια, η διακοπή λειτουργεί ως ***μηχανισμός σηματοδότησης*** και όχι ως σημείο επεξεργασίας. Το DMA συνεχίζει να γεμίζει το δεύτερο μισό του buffer, την ίδια ώρα που το main loop μπορεί να προετοιμάσει και να αποστείλει τα δεδομένα του πρώτου μισού.

4.2.6 Transfer-complete interrupt

Αντίστοιχα, όταν το DMA ολοκληρώσει τη μεταφορά ολόκληρου του buffer, δημιουργείται αντίστοιχα η λεγόμενη transfer-complete interrupt. Αυτό σημαίνει ότι το δεύτερο μισό του buffer είναι πλέον διαθέσιμο για χρήση, οπότε και τίθεται το flag `full_ready`. Το main loop, όταν διαπιστώσει ότι το υποσύστημα USB δεν είναι απασχολημένο, διαβάζει τα δεδομένα από το δεύτερο μισό των ADC buffers και τα αντιγράφει στον αντίστοιχο USB transmission buffer. Μετά το συμβάν του transfer-complete και επειδή το DMA λειτουργεί κυκλικά, η μεταφορά δεν σταματά. Αντίθετα, το DMA επιστρέφει στην αρχή του buffer και αρχίζει ξανά να γράφει στο πρώτο μισό. Έτσι επιτυγχάνεται συνεχής συλλογή δεδομένων χωρίς ανάγκη επανεκκίνησης της δειγματοληψίας από το firmware.

4.2.7 Πακετοποίηση USB

Μετά τη σηματοδότηση `half_ready` ή `full_ready`, το main loop αναλαμβάνει να μετατρέψει τα τρία ξεχωριστά ADC buffers σε ενιαίο USB packet buffer. Αυτό είναι το στάδιο της *πακετοποίησης* (packetization). Οι ADC buffers είναι αποθηκευμένοι χωριστά:

```
adc1_buf[]
adc2_buf[]
adc3_buf[]
```

Για την αποστολή όμως προς τον υπολογιστή είναι προτιμότερο κάθε χρονικό δείγμα να περιέχει μαζί τις τιμές των τριών καναλιών. Για τον λόγο αυτό το firmware δημιουργεί δομή τύπου:

```
sample.ch1
sample.ch2
sample.ch3
```

και γεμίζει έναν USB transmission buffer.
Για το πρώτο μισό του DMA buffer, η λογική είναι:

```
for (uint16_t i = 0; i < HALF_SAMPLES; i++)
{
    usb_tx_half[i].ch1 = adc1_buf[i];
    usb_tx_half[i].ch2 = adc2_buf[i];
```

```
    usb_tx_half[i].ch3 = adc3_buf[i];
}
```

Για το δεύτερο μισό, χρησιμοποιείται offset:

```
for (uint16_t i = 0; i < HALF_SAMPLES; i++)
{
    uint16_t idx = i + HALF_SAMPLES;

    usb_tx_full[i].ch1 = adc1_buf[idx];
    usb_tx_full[i].ch2 = adc2_buf[idx];
    usb_tx_full[i].ch3 = adc3_buf[idx];
}
```

Η διαδικασία αυτή μετατρέπει τα δεδομένα από μορφή “ανά κανάλι” σε μορφή “ανά πλαίσιο δείγματος (sample frame)”. Έτσι ο υπολογιστής λαμβάνει διαδοχικά πακέτα όπου κάθε στοιχείο περιέχει τις τρεις τιμές ADC που αντιστοιχούν στην ίδια χρονική θέση του buffer.

4.2.8 Έλεγχος κατάστασης USB

Πριν ξεκινήσει οποιαδήποτε USB αποστολή, το firmware ελέγχει τη μεταβλητή `usb_busy`. Η μεταβλητή αυτή δείχνει αν υπάρχει ήδη ενεργή USB μεταφορά. Αν το υποσύστημα USB είναι απασχολημένο, το main loop δεν ξεκινά νέα αποστολή. Αντίθετα, περιμένει μέχρι να ολοκληρωθεί η προηγούμενη μετάδοση. Αυτό είναι απαραίτητο επειδή το USB Device Middleware δεν πρέπει να δεχθεί νέα μετάδοση στο ίδιο τελικό σημείο όσο βρίσκεται ήδη σε εξέλιξη προηγούμενη μαζική μεταφορά. Η αποστολή γίνεται μέσω της συνάρτησης:

```
custom_class_transmit_data(
    &hUsbDeviceFS,
    (uint8_t*)usb_tx_half,
    sizeof(usb_tx_half)
);
```

ή αντίστοιχα:

```
custom_class_transmit_data(
    &hUsbDeviceFS,
    (uint8_t*)usb_tx_full,
    sizeof(usb_tx_full)
);
```

Η συνάρτηση αυτή ανήκει στο interface layer της custom USB class και εσωτερικά καλεί τη `USBD_CUSTOM_CLASS_TransmitPacket`. Το αρχείο `usbd_custom_class_if.c` λειτουργεί ως ενδιάμεσο επίπεδο μεταξύ της εφαρμογής και της USB class υλοποίησης.

4.2.9 Custom USB class transmission

Η πραγματική USB μετάδοση υλοποιείται στο αρχείο `usbd_custom_class.c`. Εκεί ορίζεται η custom USB class, οι descriptors και οι μηχανισμοί αποστολής και λήψης δεδομένων. Η συσκευή χρησιμοποιεί διεπαφή vendor-specific με class value `0xFF` και δύο μαζικά τελικά σημεία, ένα IN και ένα OUT. Η συνάρτηση `USBD_CUSTOM_CLASS_TransmitPacket` ελέγχει πρώτα αν υπάρχει ήδη μετάδοση σε εξέλιξη. Αν το εσωτερικό `TxState` είναι ενεργό, επιστρέφει `USBD_BUSY`. Αν το endpoint είναι διαθέσιμο, αποθηκεύει τον δείκτη του buffer, το μήκος της μετάδοσης και καλεί τη χαμηλού επιπέδου συνάρτηση `USBD_LL_Transmit`. Αυτό σημαίνει ότι η εφαρμογή δεν επικοινωνεί απευθείας με τους καταχωρητές (registers) USB. Αντίθετα, χρησιμοποιεί το επίπεδο αφαίρεσης (abstraction layer) της custom class, ενώ το STM USB Device Middleware αναλαμβάνει τη χαμηλού επιπέδου διαχείριση του endpoint.

4.2.10 Ολοκλήρωση USB μετάδοσης

Όταν ολοκληρωθεί η μετάδοση ενός USB packet, καλείται το Data IN callback της custom class. Στο `usbd_custom_class.c`, η συνάρτηση `USBD_CUSTOM_CLASS_DataIn` αναγνωρίζει την ολοκλήρωση της μετάδοσης και καλεί το callback του interface layer. Στο `usbd_custom_class_if.c`, το transmit complete callback μηδενίζει τη μεταβλητή:

```
usb_busy = 0;
```

Με αυτόν τον τρόπο ενημερώνεται το main loop ότι μπορεί να ξεκινήσει η αποστολή του επόμενου packet. Η μεταβλητή `usb_busy` λειτουργεί επομένως ως απλός μηχανισμός συγχρονισμού μεταξύ application layer και USB transport layer και δεν αποτελεί queue, αλλά προστατεύει από την επικάλυψη δύο διαδοχικών μεταδόσεων.

4.2.11 Παρατηρήσεις για cache coherency

Στον αρχικό κώδικα υπάρχουν κλήσεις:

```
SCB_CleanDCache_by_Addr(...)
```

Οι κλήσεις αυτές σχετίζονται με τη λειτουργία της D-Cache του Cortex-M7. Όταν δεδομένα που έχουν γραφτεί από τον CPU πρόκειται να σταλούν μέσω DMA ή περιφερειακού USB, ενδέχεται να απαιτείται καθάρισμα της cache ώστε τα πραγματικά δεδομένα να βρίσκονται στη RAM και όχι μόνο στην cache. Το θέμα αυτό είναι ιδιαίτερα σημαντικό στον STM32H7, επειδή η ύπαρξη cache μπορεί να προκαλέσει ασυμφωνία μεταξύ των δεδομένων που βλέπει ο CPU και των δεδομένων που προσπελαύνουν τα περιφερειακά. Αξίζει να αναφερθεί ότι, στην παρούσα εργασία, το firmware λαμβάνει υπόψη αυτά τα ζητήματα ακόμη και αν στην τελική διαμόρφωση οι συγκεκριμένες κλήσεις δεν χρησιμοποιούνται ενεργά λόγω επιλογής κατάλληλων memory regions ή λόγω της συγκεκριμένης συμπεριφοράς του USB stack.

4.2.12 Συμπέρασμα

Η ροή δεδομένων του firmware βασίζεται σε μία απλή αλλά αποδοτική αρχιτεκτονική παραγωγού-καταναλωτή. Το υποσύστημα hardware acquisition, αποτελούμενο από ADC και DMA, λειτουργεί ως παραγωγός δεδομένων, ενώ ο κύριος βρόχος και το υποσύστημα USB λειτουργούν ως καταναλωτής και μηχανισμός μεταφοράς προς τον υπολογιστή. Η επιλογή double buffering αντί για queue μειώνει την πολυπλοκότητα του firmware και επιτρέπει ντετερμινιστική λειτουργία. Οι διακοπές half-transfer

και transfer-complete χρησιμοποιούνται μόνο για σηματοδότηση, ενώ η πραγματική επεξεργασία γίνεται στο main loop. Η custom USB class με μαζικό τελικό σημείο IN αναλαμβάνει την αποστολή των πακέτων προς τον host, ενώ το `usb_busy` flag εξασφαλίζει ότι δεν ξεκινά νέα μετάδοση πριν ολοκληρωθεί η προηγούμενη.

Κεφάλαιο 5ο: PC εφαρμογή σε Python

5.1 Python Stack και Εφαρμογή Λήψης Δεδομένων

Η εφαρμογή του υπολογιστή αναπτύχθηκε σε Python και αποτελεί το τελικό επίπεδο του συστήματος συλλογής δεδομένων. Ο ρόλος της είναι να επικοινωνεί με τη USB συσκευή, να λαμβάνει τα πακέτα μετρήσεων, να τα αποκωδικοποιεί σε τιμές ADC και να τα εμφανίζει σε πραγματικό χρόνο. Η εφαρμογή αυτή δεν αντικαθιστά ένα πλήρες περιβάλλον επιστημονικής ανάλυσης, αλλά λειτουργεί ως βασικό εργαλείο λήψης, ελέγχου και αρχικής απεικόνισης των δεδομένων.
Η επιλογή της γλώσσας Python έγινε επειδή προσφέρει γρήγορη ανάπτυξη, εύκολη πρόσβαση σε βιβλιοθήκες USB, ισχυρή υποστήριξη αριθμητικών δεδομένων και άμεση δυνατότητα απεικόνισης. Επιπλέον πλεονέκτημα της Python είναι ότι συνδέεται εύκολα με περιβάλλοντα όπως το Jupyter Notebook, όπου μπορούν να γίνουν πιο σύνθετες διαδικασίες ανάλυσης μετά τη συλλογή των δεδομένων. Έτσι, η εφαρμογή του υπολογιστή χωρίζεται νοητά σε δύο επίπεδα. Το πρώτο είναι το πρόγραμμα λήψης και απεικόνισης σε πραγματικό χρόνο, ενώ το δεύτερο είναι το περιβάλλον μεταγενέστερης επεξεργασίας, το οποίο μπορεί να βασίζεται σε Jupyter Notebook, NumPy, Matplotlib ή άλλες βιβλιοθήκες. Για την παρούσα εργασία δεν ενσωματώθηκε πλήρης επεξεργασία σήματος μέσα στο πρόγραμμα λήψης. Η σχεδιαστική επιλογή ήταν τα δεδομένα να μπορούν να ληφθούν, να απεικονιστούν και στη συνέχεια να χρησιμοποιηθούν σε ξεχωριστό περιβάλλον ανάλυσης. Αυτό δίνει μεγαλύτερη ευελιξία, καθώς η επεξεργασία μπορεί να μεταβάλλεται ανάλογα με το πείραμα χωρίς να απαιτούνται αλλαγές στο firmware ή στον βασικό κώδικα επικοινωνίας.

5.2 Περιβάλλον εκτέλεσης και φορητότητα

Για την ανάπτυξη της εφαρμογής χρησιμοποιήθηκε το οικοσύστημα Python με εξωτερικές βιβλιοθήκες για USB επικοινωνία, αριθμητικούς υπολογισμούς και γραφική απεικόνιση. Σημαντική πρακτική επιλογή εκτιμώ ότι ήταν η χρήση τοπικού εικονικού περιβάλλοντος (virtual environment), δηλαδή φακέλου `.venv`, μέσα στον κατάλογο του project γιατί το virtual environment επιτρέπει την απομόνωση των βιβλιοθηκών της εφαρμογής από το υπόλοιπο σύστημα. Με αυτόν τον τρόπο, οι εκδόσεις των βιβλιοθηκών που απαιτούνται για το πρόγραμμα παραμένουν σταθερές και δεν επηρεάζονται από άλλες εγκαταστάσεις Python του υπολογιστή. Αυτό είναι ιδιαίτερα χρήσιμο σε εργαστηριακά projects, όπου το πρόγραμμα πρέπει να μπορεί να μεταφερθεί σε διαφορετικό υπολογιστή χωρίς να δημιουργούνται προβλήματα εξαρτήσεων. Η τυπική δομή του φακέλου μπορεί να είναι:

```
pc_software/
│
├── .venv/
├── main.py
├── libusb-1.0.dll
└── requirements.txt
```

Το αρχείο `libusb-1.0.dll` τοποθετείται στον ίδιο φάκελο με το Python script, ώστε το πρόγραμμα να μπορεί να το εντοπίζει άμεσα. Η επιλογή αυτή κάνει την εφαρμογή πιο φορητή, επειδή δεν απαιτείται απαραίτητα καθολική εγκατάσταση της βιβλιοθήκης libusb στο σύστημα. Ο κώδικας χρησιμοποιεί τη διαδρομή του τρέχοντος αρχείου και αναζητά το DLL τοπικά:

```
dll_path = pathlib.Path(__file__).parent / "libusb-1.0.dll"
```

Με αυτόν τον τρόπο, το πρόγραμμα μπορεί να εκτελεστεί από διαφορετικούς υπολογιστές με μικρότερη πιθανότητα σφαλμάτων λόγω ελλειπόντων διαδρομών συστήματος (system paths).

5.2.1 Πρόσβαση στη USB συσκευή

Η επικοινωνία με τη συσκευή γίνεται μέσω της βιβλιοθήκης PyUSB, η οποία παρέχει Python interface για πρόσβαση σε USB συσκευές. Στα Windows, η PyUSB χρησιμοποιεί backend βασισμένο στο `libusb-1.0.dll`. Η συσκευή αναγνωρίζεται με βάση το Vendor ID και το Product ID:

```
VID = 0x0483
PID = 0x0002
```

Οι τιμές αυτές πρέπει να αντιστοιχούν στα descriptors του firmware. Το VID `0x0483` αντιστοιχεί στη STMicroelectronics, ενώ το PID χρησιμοποιείται για την αναγνώριση της συγκεκριμένης συσκευής από το host software. Η συνάρτηση `find_device()` είναι υπεύθυνη για την ανίχνευση της USB συσκευής, την επιλογή του backend και τον εντοπισμό του IN endpoint. Αρχικά φορτώνεται το libusb backend:

```
backend = usb.backend.libusb1.get_backend(
    find_library=lambda x: str(dll_path)
)
```

Στη συνέχεια γίνεται αναζήτηση της συσκευής:

```
dev = usb.core.find(
    idVendor=VID,
    idProduct=PID,
    backend=backend
)
```

Αν η συσκευή δεν βρεθεί, το πρόγραμμα σταματά με μήνυμα σφάλματος. Αυτό είναι χρήσιμο κατά το debugging, επειδή ξεχωρίζει άμεσα τα προβλήματα σύνδεσης ή driver, από τα προβλήματα αποκωδικοποίησης δεδομένων.
Μετά τον εντοπισμό της συσκευής, γίνεται επιλογή configuration και ανάγνωση του active interface:

```
dev.set_configuration()
cfg = dev.get_active_configuration()
intf = cfg[(0, 0)]
```

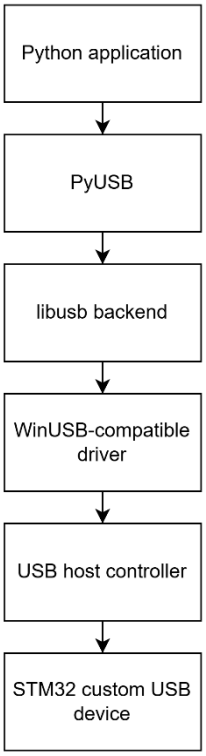
Τέλος, το πρόγραμμα αναζητά το endpoint εισόδου, δηλαδή το τελικό σημείο από το οποίο ο υπολογιστής θα λαμβάνει δεδομένα από τη συσκευή:

```
ep_in = usb.util.find_descriptor(
    intf,
    custom_match=lambda e:
    usb.util.endpoint_direction(
        e.bEndpointAddress
    ) == usb.util.ENDPOINT_IN
)
```

Το endpoint αυτό αντιστοιχεί στο μαζικό τερματικό εισόδου (Bulk IN endpoint) της ειδικής κλάσης USB του firmware.

5.2.2 WinUSB, libusb και Python bindings

Στο επίπεδο του λειτουργικού συστήματος, η συσκευή USB δεν εμφανίζεται ως σειριακή θύρα, όπως συνέβαινε στην αρχική CDC υλοποίηση. Αντίθετα, χρησιμοποιείται custom USB class με bulk endpoints. Για τον λόγο αυτό απαιτείται κατάλληλος user-space driver ή USB backend.
Στα Windows, μία πρακτική λύση είναι η χρήση WinUSB-compatible driver. Το WinUSB επιτρέπει σε εφαρμογές user-space να επικοινωνούν με USB συσκευές χωρίς την ανάπτυξη custom kernel driver. Πάνω από αυτό το επίπεδο μπορεί να χρησιμοποιηθεί το libusb, το οποίο παρέχει ενιαίο API πρόσβασης σε USB endpoints. Η PyUSB αποτελεί Python binding που χρησιμοποιεί αυτό το backend και επιτρέπει στον κώδικα Python να εκτελεί USB transfers χωρίς άμεση χρήση WinUSB API ή C κώδικα.



Σχήμα 5.1: Software stack του υπολογιστή

Η προσέγγιση αυτή αφενός είναι αρκετά απλή για ανάπτυξη και debugging, αφετέρου επιτρέπει πρόσβαση στα bulk endpoints της συσκευής. Με αυτόν τον τρόπο αποφεύγεται η ανάγκη ανάπτυξης custom Windows kernel driver, κάτι που θα αύξανε σημαντικά την πολυπλοκότητα της εργασίας.

5.2.3 Ανάγνωση δεδομένων από το USB endpoint

Η βασική λειτουργία της εφαρμογής εκτελείται μέσα σε έναν ατέρμονα βρόχο. Σε κάθε επανάληψη, το πρόγραμμα προσπαθεί να διαβάσει ένα πακέτο από το Bulk IN endpoint:

```
data = ep_in.read(USB_PACKET_SIZE, timeout=100)
```

Το μέγεθος ανάγνωσης έχει οριστεί σε 64 bytes:

```
USB_PACKET_SIZE = 64
```

Η τιμή αυτή αντιστοιχεί στο μέγιστο μέγεθος πακέτου ενός μαζικού τελικού σημείου USB Full Speed. Με αυτόν τον τρόπο, κάθε ανάγνωση από το host ζητά ένα πλήρες πακέτο USB από τη συσκευή. Η ανάγνωση διαθέτει timeout 100 ms. Αν στο διάστημα αυτό δεν ληφθούν δεδομένα, δημιουργείται εξαίρεση τύπου `USBTimeoutError`, η οποία αγνοείται λειτουργικά και απλώς επιτρέπει στο πρόγραμμα να συνεχίσει να ανανεώνει το plot:

```
except usb.core.USBTimeoutError:
    plt.pause(0.001)
```

Αυτό είναι σημαντικό επειδή η εφαρμογή δεν πρέπει να τερματίζεται κάθε φορά που προσωρινά δεν υπάρχουν δεδομένα. Το timeout είναι φυσιολογικό γεγονός σε εφαρμογές USB, ειδικά όταν η συσκευή δεν στέλνει συνεχώς δεδομένα ή όταν βρίσκεται σε κατάσταση αναμονής.

5.2.4 Ασύγχρονη συμπεριφορά μέσω polling loop

Η υλοποίηση δεν χρησιμοποιεί πραγματικό asynchronous I/O με callbacks ή ξεχωριστά νήματα (threads). Παρ' όλα αυτά, η εφαρμογή έχει ασύγχρονη συμπεριφορά σε επίπεδο λειτουργίας, επειδή δεν περιμένει προκαθορισμένο αριθμό δειγμάτων ούτε βασίζεται σε φράζουσα (blocking) ροή χωρίς χρονικό όριο (timeout). Ο βρόχος ανάγνωσης λειτουργεί ως polling loop με μικρό timeout. Το πρόγραμμα ζητά δεδομένα από το endpoint, επεξεργάζεται όσα bytes έχουν ληφθεί και στη συνέχεια ανανεώνει την απεικόνιση. Αν δεν υπάρχουν δεδομένα, δεν τερματίζει, αλλά συνεχίζει την εκτέλεση.
Η παραπάνω σχεδίαση είναι απλή και επαρκής για την παρούσα φάση της εργασίας. Σε μελλοντική επέκταση, η ανάγνωση USB θα μπορούσε να μεταφερθεί σε ξεχωριστό νήμα ή σε πραγματικό ασύγχρονο μηχανισμό, ώστε το plotting και το acquisition να αποσυνδεθούν πλήρως. Ωστόσο, για την αρχική επιβεβαίωση της λειτουργίας του συστήματος, ο απλός polling βρόχος είναι πιο κατανοητός, πιο εύκολος στο debugging και αρκετά αποτελεσματικός.

5.2.5 Packet decoding

Το firmware αποστέλλει τα δείγματα σε δυαδική μορφή. Κάθε sample frame αποτελείται από τρεις τιμές ADC, μία για κάθε κανάλι. Κάθε τιμή είναι 16-bit unsigned integer και μεταδίδεται σε little-endian μορφή. Επομένως, κάθε sample frame έχει μέγεθος:

```
SAMPLE_SIZE = 6
```

δηλαδή:

```
2 bytes για ADC1
2 bytes για ADC2
2 bytes για ADC3
```

Η διάταξη κάθε sample frame είναι:

```
Byte 0-1: ADC1
Byte 2-3: ADC2
Byte 4-5: ADC3
```

Η αποκωδικοποίηση γίνεται με τη μέθοδο `int.from_bytes()`:

```
ch1 = int.from_bytes(sample[0:2], 'little')
ch2 = int.from_bytes(sample[2:4], 'little')
ch3 = int.from_bytes(sample[4:6], 'little')
```

Με αυτόν τον τρόπο τα raw bytes μετατρέπονται σε ακέραιες τιμές από 0 έως 65535, που αντιστοιχούν στο εύρος ενός 16-bit ADC. Επειδή η USB μεταφορά δεν εγγυάται ότι κάθε κλήση `read()` θα περιέχει ακριβώς ακέραιο αριθμό ολοκληρωμένων sample frames, χρησιμοποιείται ενδιάμεσος byte buffer:

```
buffer = bytearray()
```

Κάθε νέο πακέτο USB προστίθεται στο buffer:

```
buffer.extend(data)
```

Στη συνέχεια, όσο υπάρχουν τουλάχιστον 6 bytes διαθέσιμα, το πρόγραμμα αφαιρεί ένα sample frame και το αποκωδικοποιεί:

```
while len(buffer) >= SAMPLE_SIZE:
    sample = buffer[:SAMPLE_SIZE]
    del buffer[:SAMPLE_SIZE]
```

Η τεχνική αυτή κάνει τον αποκωδικοποιητή (decoder) πιο ανθεκτικό, επειδή δεν βασίζεται απόλυτα στα όρια των USB packets. Το πρόγραμμα αντιμετωπίζει το USB ως συνεχή ροή bytes και εξάγει δείγματα όταν υπάρχουν αρκετά δεδομένα.

5.2.6 Buffering στην εφαρμογή Python

Η εφαρμογή χρησιμοποιεί δύο διαφορετικά επίπεδα buffering. Το πρώτο είναι το byte-level buffer, δηλαδή το `bytearray`, το οποίο συγκεντρώνει τα bytes από το USB μέχρι να σχηματιστεί πλήρες sample frame. Το δεύτερο είναι το plotting buffer, το οποίο αποθηκεύει τις πιο πρόσφατες τιμές κάθε καναλιού για εμφάνιση στο γράφημα. Για την απεικόνιση χρησιμοποιούνται τρία `deque` αντικείμενα:

```
ch1_buf = deque([0] * history, maxlen=history)
ch2_buf = deque([0] * history, maxlen=history)
ch3_buf = deque([0] * history, maxlen=history)
```

Το μέγεθος του ιστορικού έχει οριστεί σε:

```
history = 512
```

Κάθε deque κρατά μόνο τα τελευταία 512 δείγματα του αντίστοιχου καναλιού. Όταν προστίθεται νέα τιμή και το buffer είναι ήδη γεμάτο, η παλαιότερη τιμή αφαιρείται αυτόματα. Αυτό καθιστά το plotting απλό και αποδοτικό, καθώς το γράφημα δείχνει πάντα ένα κυλιόμενο παράθυρο των τελευταίων δειγμάτων. Η χρήση `deque` είναι κατάλληλη για real-time απεικόνιση, επειδή υποστηρίζει γρήγορη εισαγωγή νέων στοιχείων χωρίς ανάγκη μετακίνησης ολόκληρου πίνακα κάθε φορά.

5.2.7 Real-time plotting

Η απεικόνιση των δεδομένων γίνεται με τη βιβλιοθήκη Matplotlib. Η εφαρμογή ενεργοποιεί interactive mode:

```
plt.ion()
```

και δημιουργεί ένα γράφημα με τρεις γραμμές, μία για κάθε ADC κανάλι:

```
line1, = ax.plot(x, np.zeros(history), label='ADC1')
line2, = ax.plot(x, np.zeros(history), label='ADC2')
line3, = ax.plot(x, np.zeros(history), label='ADC3')
```

Ο άξονας y ορίζεται στο πλήρες εύρος 16-bit τιμών:

```
ax.set_ylim(0, 65535)
```

Σε κάθε επανάληψη του βρόχου, αφού αποκωδικοποιηθούν τα νέα δείγματα , ενημερώνονται τα δεδομένα των γραμμών:

```
line1.set_ydata(ch1_buf)
line2.set_ydata(ch2_buf)
line3.set_ydata(ch3_buf)
```

και στη συνέχεια γίνεται ανανέωση του γραφήματος:

```
fig.canvas.draw_idle()
plt.pause(0.001)
```

Η χρήση του `draw_idle()` αντί για πλήρη επανασχεδίαση από την αρχή μειώνει το φόρτο και επιτρέπει πιο ομαλή απεικόνιση. Το `plt.pause()` επιτρέπει στο event loop του Matplotlib να επεξεργαστεί τα GUI events και να ενημερώσει το παράθυρο.
Η συγκεκριμένη προσέγγιση είναι κατάλληλη για αρχική παρακολούθηση των σημάτων και για debugging της μεταφοράς δεδομένων. Για πολύ υψηλούς ρυθμούς λήψης, η απεικόνιση σε πραγματικό χρόνο μπορεί να αποτελέσει bottleneck, καθώς η Matplotlib δεν έχει σχεδιαστεί για πολύ υψηλό frame rate. Σε τέτοιες περιπτώσεις θα ήταν προτιμότερο να γίνεται το λεγόμενο decimation των δεδομένων πριν την απεικόνιση ή να χρησιμοποιηθεί πιο αποδοτική βιβλιοθήκη γραφικών.

5.2.8 Terminal logging

Η εφαρμογή υποστηρίζει προαιρετική εκτύπωση των τιμών στο τερματικό (terminal) μέσω της παραμέτρου:

```
--log
```

Η παράμετρος αυτή υλοποιείται με τη βιβλιοθήκη `argparse`:

```
parser.add_argument(
    "--log",
    action="store_true",
    help="Enable terminal logging"
)
```

Όταν ενεργοποιηθεί, το πρόγραμμα εκτυπώνει κάθε sample frame:

```
if args.log:
    print(ch1, ch2, ch3)
```

Η λειτουργία αυτή είναι χρήσιμη για debugging, επειδή επιτρέπει τον άμεσο έλεγχο των αριθμητικών τιμών χωρίς να απαιτείται γραφική απεικόνιση. Ωστόσο, η συνεχής εκτύπωση στο τερματικό είναι εξαιρετικά αργή σε σχέση με τον ρυθμό λήψης δεδομένων και μπορεί να μειώσει σημαντικά την απόδοση του προγράμματος. Για τον λόγο αυτό η λειτουργία logging είναι προαιρετική και απενεργοποιημένη από προεπιλογή.

5.2.9 Σχέση με Jupyter Notebook

Η εφαρμογή Python που παρουσιάζεται εδώ έχει ως κύριο σκοπό τη λήψη και την αρχική απεικόνιση των δεδομένων. Η πιο σύνθετη επεξεργασία των σημάτων δεν υλοποιείται μέσα στο ίδιο πρόγραμμα, αλλά προορίζεται να γίνεται σε μεταγενέστερο στάδιο, πιθανώς μέσα σε Jupyter Notebook.
Η λογική αυτή έχει αρκετά πλεονεκτήματα. Καταρχήν, διαχωρίζει τη συλλογή από την ανάλυση· το πρόγραμμα λήψης παραμένει απλό και σταθερό, ενώ οι αλγόριθμοι ανάλυσης μπορούν να αλλάζουν ελεύθερα ανάλογα με το πείραμα. Δεύτερον, το Jupyter Notebook επιτρέπει διαδραστική εργασία με τα δεδομένα, δημιουργία γραφημάτων, εφαρμογή φίλτρων, υπολογισμό στατιστικών μεγεθών και τεκμηρίωση των αποτελεσμάτων στον ίδιο χώρο.
Σε μελλοντική επέκταση, η εφαρμογή λήψης θα μπορούσε να αποθηκεύει τα δεδομένα σε αρχείο, όπως CSV, NumPy `.npy` ή binary format. Το αρχείο αυτό θα μπορούσε στη συνέχεια να εισάγεται σε notebook για περαιτέρω επεξεργασία.

5.2.10 Περιορισμοί της παρούσας εφαρμογής

Η εφαρμογή στο πλαίσιο της παρούσας εργασίας υπήρξε σκόπιμα απλή και επικεντρώνεται στη βασική λειτουργία λήψης και απεικόνισης. Δεν περιλαμβάνει ακόμη πλήρη μηχανισμό αποθήκευσης μεγάλου όγκου δεδομένων, έλεγχο απώλειας πακέτων, μετρητές ακολουθίας (sequence counters) ή αυτόματο επανασυγχρονισμό (resynchronization) σε περίπτωση απώλειας byte alignment.
Επίσης, επειδή το USB stream αποκωδικοποιείται ως συνεχής ακολουθία από 6-byte sample frames, το πρόγραμμα θεωρεί ότι η στοίχιση των πακέτων παραμένει σωστή. Ωστόσο σε πιο ολοκληρωμένη έκδοση του πρωτοκόλλου θα ήταν χρήσιμο να προστεθεί header, sample counter ή sync word, ώστε το PC software να μπορεί να εντοπίζει την αρχή πακέτου και να ανιχνεύει πιθανές απώλειες.
Ένας ακόμη περιορισμός της παρούσας εφαρμογής είναι ότι η Matplotlib μπορεί να επιβαρύνει σημαντικά την απόδοση όταν ο ρυθμός εισόδου αυξηθεί. Για χαμηλότερα sample rates είναι επαρκής, αλλά για υψηλότερους ρυθμούς θα ήταν προτιμότερο το plotting να γίνεται σε χαμηλότερο ρυθμό ανανέωσης από ό,τι η συλλογή.

5.2.11 Συμπέρασμα

Το Python software αποτελεί το host-side τμήμα του συστήματος συλλογής και επιτρέπει την επικοινωνία με τη custom USB συσκευή μέσω PyUSB και libusb backend. Η εφαρμογή εντοπίζει τη συσκευή βάσει VID/PID, ανοίγει το Bulk IN endpoint, λαμβάνει πακέτα USB, τα οποία αποκωδικοποιεί σε τριάδες 16-bit ADC τιμών, και εμφανίζει τα δεδομένα σε πραγματικό χρόνο μέσω Matplotlib.
Η χρήση `.venv` και τοπικού `libusb-1.0.dll` βελτιώνει τη φορητότητα του περιβάλλοντος ανάπτυξης, ενώ η απλή αρχιτεκτονική του κώδικα επιτρέπει εύκολο debugging και μελλοντική επέκταση. Η επεξεργασία σημάτων δεν ενσωματώνεται άμεσα στο πρόγραμμα λήψης, αλλά προβλέπεται να γίνεται σε ξεχωριστό περιβάλλον, όπως το Jupyter Notebook, όπου τα δεδομένα μπορούν να αναλυθούν με μεγαλύτερη ευελιξία.

Κεφάλαιο 6ο: Συμπεράσματα και Μελλοντική Εργασία

Η παρούσα διπλωματική εργασία είχε ως στόχο τη σχεδίαση και υλοποίηση ενός συστήματος συλλογής δεδομένων υψηλής ταχύτητας με τρία ανεξάρτητα αναλογικά κανάλια μέτρησης, καθώς και την ανάπτυξη λογισμικού υπολογιστή για τη λήψη και απεικόνιση των δεδομένων. Για την υλοποίηση του συστήματος επιλέχθηκε ο μικροελεγκτής STM32H743ZI, ο οποίος διαθέτει πυρήνα ARM Cortex-M7 υψηλής απόδοσης, τρεις ενσωματωμένους μετατροπείς αναλογικού σε ψηφιακό σήμα και προηγμένους μηχανισμούς DMA και USB επικοινωνίας.

Κατά τη διάρκεια της εργασίας μελετήθηκαν και αξιοποιήθηκαν τεχνολογίες που χρησιμοποιούνται ευρέως σε σύγχρονα συστήματα απόκτησης δεδομένων, όπως οι μετατροπείς ADC, οι μηχανισμοί Direct Memory Access, η επικοινωνία USB και η ανάπτυξη λογισμικού σε γλώσσα Python. Το τελικό σύστημα που αναπτύχθηκε είναι σε θέση να συλλέγει δεδομένα από τρία αναλογικά κανάλια και να τα μεταφέρει στον υπολογιστή μέσω USB, επιτρέποντας την παρακολούθηση και επεξεργασία των μετρήσεων.

Ένα από τα σημαντικότερα αποτελέσματα της εργασίας ήταν η επιτυχής υλοποίηση ενός πλήρους acquisition pipeline, στο οποίο η συλλογή των δειγμάτων πραγματοποιείται από τους ADC, η μεταφορά τους προς τη μνήμη γίνεται μέσω DMA σε circular double-buffer mode και η αποστολή τους προς τον υπολογιστή πραγματοποιείται μέσω custom USB class με bulk endpoints. Η αρχιτεκτονική αυτή επέτρεψε την αποδοτική εκμετάλλευση των δυνατοτήτων του μικροελεγκτή και την ελαχιστοποίηση της επιβάρυνσης του επεξεργαστή κατά τη διάρκεια της συλλογής δεδομένων.

Η χρήση των ADC1 και ADC2 σε συγχρονισμένο dual mode επέτρεψε τη σχεδόν ταυτόχρονη δειγματοληψία δύο καναλιών, ενώ ο ADC3 χρησιμοποιήθηκε ως τρίτο ανεξάρτητο κανάλι μέτρησης. Η χρήση DMA σε συνδυασμό με τους half-transfer και transfer-complete μηχανισμούς επέτρεψε τη συνεχή λειτουργία του συστήματος χωρίς απώλειες δεδομένων για τους ρυθμούς δειγματοληψίας που εξετάστηκαν. Παράλληλα, η ανάπτυξη custom USB Device Class επέτρεψε την άμεση ανταλλαγή δυαδικών δεδομένων με τον υπολογιστή χωρίς την επιβάρυνση που συνεπάγονται πρωτόκολλα ανώτερου επιπέδου ή η χρήση εικονικής σειριακής θύρας.

Στην πλευρά του υπολογιστή αναπτύχθηκε εφαρμογή σε Python η οποία αξιοποιεί τις βιβλιοθήκες PyUSB και libusb για την επικοινωνία με τη συσκευή. Η εφαρμογή είναι σε θέση να λαμβάνει τα δεδομένα, να τα αποκωδικοποιεί και να τα απεικονίζει σε πραγματικό χρόνο. Παρότι η επεξεργασία σημάτων δεν αποτέλεσε κύριο αντικείμενο της εργασίας, η χρήση της Python παρέχει ένα ιδιαίτερα ευέλικτο περιβάλλον για μελλοντική ανάλυση των δεδομένων μέσω εργαλείων όπως το NumPy, το SciPy και τα Jupyter Notebooks.

Κατά την ανάπτυξη του συστήματος εμφανίστηκαν αρκετές τεχνικές προκλήσεις. Μία από τις σημαντικότερες ήταν η διαχείριση της πολύπλοκης αρχιτεκτονικής μνήμης του STM32H743. Η ύπαρξη πολλαπλών memory domains και διαφορετικών DMA controllers απαίτησε προσεκτική μελέτη των περιοχών μνήμης και τροποποιήσεις στα linker scripts του έργου. Ιδιαίτερη προσοχή απαιτήθηκε στη χρήση του BDMA και στην τοποθέτηση συγκεκριμένων buffers σε κατάλληλες περιοχές μνήμης ώστε να εξασφαλιστεί η σωστή λειτουργία του συστήματος. Παράλληλα, η ύπαρξη cache στον πυρήνα Cortex-M7 δημιούργησε ζητήματα coherency μεταξύ CPU και DMA, τα οποία έπρεπε να ληφθούν υπόψη κατά τη σχεδίαση της εφαρμογής.

Παρότι το τελικό σύστημα λειτούργησε επιτυχώς, υπάρχουν ορισμένοι περιορισμοί που επηρεάζουν τις μέγιστες επιδόσεις του. Ο σημαντικότερος περιορισμός σχετίζεται με τη χρήση USB Full Speed. Το USB Full Speed προσφέρει θεωρητικό ρυθμό μετάδοσης 12 Mbit/s, ο οποίος στην πράξη είναι ακόμη μικρότερος λόγω του πρωτοκόλλου και των σχετικών επικεφαλίδων μεταφοράς. Αν ληφθεί υπόψη ότι κάθε δείγμα αποτελείται από τρεις τιμές 16 bit, ο όγκος δεδομένων που παράγεται από το σύστημα μπορεί εύκολα να υπερβεί τις δυνατότητες συνεχούς μεταφοράς του USB Full Speed σε υψηλούς

ρυθμούς δειγματοληψίας. Για τον λόγο αυτό η απευθείας συνεχής μετάδοση δεδομένων είναι εφικτή μόνο για χαμηλότερα sample rates, ενώ για υψηλότερους ρυθμούς απαιτείται προσωρινή αποθήκευση των δεδομένων στη μνήμη του μικροελεγκτή.

Ένας ακόμη περιορισμός προέρχεται από τη χρήση των εσωτερικών ADC του μικροελεγκτή. Παρότι οι σύγχρονοι ενσωματωμένοι ADC προσφέρουν πολύ καλές επιδόσεις και αποτελούν οικονομικά αποδοτική λύση, εξακολουθούν να υπολείπονται εξειδικευμένων εξωτερικών μετατροπέων όσον αφορά την ακρίβεια, το effective number of bits (ENOB), τον θόρυβο και τη γραμμικότητα. Οι εσωτερικοί ADC επηρεάζονται επίσης από τη λειτουργία των υπόλοιπων υποσυστημάτων του μικροελεγκτή, όπως τα ρολόγια, οι DMA μεταφορές και το USB peripheral, γεγονός που μπορεί να αυξήσει το επίπεδο θορύβου στις μετρήσεις.

Μία σημαντική μελλοντική επέκταση του συστήματος θα ήταν η μετάβαση σε USB High Speed. Με τη χρήση κατάλληλου μικροελεγκτή ή εξωτερικού USB High Speed PHY θα ήταν δυνατή η επίτευξη θεωρητικού ρυθμού μετάδοσης έως 480 Mbit/s. Ένα τέτοιο bandwidth θα επέτρεπε τη συνεχή μετάδοση σημαντικά μεγαλύτερου όγκου δεδομένων και θα καθιστούσε εφικτή τη λειτουργία του συστήματος σε πολύ υψηλότερους ρυθμούς δειγματοληψίας χωρίς την ανάγκη εκτεταμένης προσωρινής αποθήκευσης.

Μία δεύτερη προφανής επέκταση αφορά τη χρήση εξωτερικών ADC υψηλότερων επιδόσεων. Εξειδικευμένοι μετατροπείς 16 ή 18 bit με βελτιωμένα χαρακτηριστικά θορύβου και μεγαλύτερη ακρίβεια θα μπορούσαν να βελτιώσουν σημαντικά την ποιότητα των μετρήσεων. Η επικοινωνία τους με τον μικροελεγκτή θα μπορούσε να πραγματοποιηθεί μέσω SPI, QSPI ή παράλληλων διεπαφών υψηλής ταχύτητας, επιτρέποντας τη δημιουργία ενός συστήματος πιο κοντά στις δυνατότητες επαγγελματικών οργάνων μέτρησης.

Επιπλέον, ιδιαίτερο ενδιαφέρον παρουσιάζει η προσθήκη εξωτερικής SDRAM ή SRAM μεγάλης χωρητικότητας. Η χρήση εξωτερικής μνήμης θα επέτρεπε την προσωρινή αποθήκευση σημαντικά μεγαλύτερου αριθμού δειγμάτων πριν από τη μεταφορά τους στον υπολογιστή. Με τον τρόπο αυτό θα ήταν δυνατή η καταγραφή πολύ γρήγορων φαινομένων για μικρό χρονικό διάστημα ακόμη και όταν το διαθέσιμο communication bandwidth δεν επαρκεί για πραγματικό streaming. Η προσέγγιση αυτή χρησιμοποιείται ευρέως σε παλμογράφους και συστήματα data acquisition υψηλών επιδόσεων.

Μία ακόμη πιθανή επέκταση αφορά την ενσωμάτωση λειτουργιών ανάλυσης δεδομένων στην εφαρμογή του υπολογιστή. Θα μπορούσαν να προστεθούν δυνατότητες αποθήκευσης σε αρχεία, υπολογισμού στατιστικών μεγεθών, ψηφιακών φίλτρων, μετασχηματισμών Fourier και αυτόματης εξαγωγής χαρακτηριστικών των σημάτων. Παράλληλα, η χρήση Jupyter Notebook θα μπορούσε να αποτελέσει μία ολοκληρωμένη πλατφόρμα πειραματισμού και ανάλυσης για διαφορετικά είδη μετρήσεων.

Συνολικά, η εργασία απέδειξε ότι οι δυνατότητες ενός σύγχρονου μικροελεγκτή όπως ο STM32H743 είναι επαρκείς για την υλοποίηση ενός πολυκαναλικού συστήματος συλλογής δεδομένων υψηλής ταχύτητας με σχετικά μικρό κόστος και περιορισμένη πολυπλοκότητα υλικού. Παράλληλα, αναδείχθηκαν οι τεχνικές προκλήσεις που εμφανίζονται σε συστήματα υψηλού throughput, όπως η διαχείριση μνήμης, οι DMA μεταφορές και οι περιορισμοί των διαύλων επικοινωνίας. Το τελικό αποτέλεσμα αποτελεί μία πλήρως λειτουργική βάση πάνω στην οποία μπορούν να αναπτυχθούν πιο προηγμένα συστήματα acquisition στο μέλλον.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Βιβλία

[1] The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, California, USA, 1997.

[2] The Definitive Guide to ARM Cortex-M7 and Cortex-M7 Processors, Joseph Yiu, *The Definitive Guide to ARM Cortex-M7 and Cortex-M7 Processors*, 3rd Edition, Newnes, Oxford, United Kingdom, 2018.

[3] Mastering STM32, Carmine Noviello, *Mastering STM32*, 2nd Edition, Leanpub, 2022.

[4] USB Complete: The Developer's Guide, Jan Axelson, *USB Complete: The Developer's Guide*, 5th Edition, Lakeview Research LLC, Madison, Wisconsin, USA, 2015.

Application Note

[5] STMicroelectronics, "AN4891 Application note STM32H72x, STM32H73x, and single-core STM32H74x/75x system architecture and performance," Application note AN4891, Jul 2025

Data Sheet

[6] STMicroelectronics, "STM32H742xI/G STM32H743xI/G 32-bit Arm® Cortex®-M7 480MHz MCUs, up to 2MB flash, up to 1MB RAM, 46 com. and analog interfaces," STM32H743 datasheet, Jan. 2026.

Manuals

[7] STMicroelectronics, "RM0433 Reference manual STM32H742, STM32H743/753 and STM32H750 Value line advanced Arm®-based 32-bit MCUs," STM32H743 Reference manual, Jan. 2023.

[8] STMicroelectronics, "PM0253 Programming manual STM32F7 series and STM32H7 series Cortex®-M7 processor programming manual," STM32H7 Programming manual, May. 2026.

Internet Site

[9] PyUSB, PyUSB Development Team, *PyUSB Documentation*, Open Source Software Documentation.

[10] NumPy, NumPy Developers, *NumPy Documentation*, Open Source Software Documentation.

[11] Matplotlib, Matplotlib Development Team, *Matplotlib Documentation*, Open Source Software Documentation.

[12] libusb, libusb Project Contributors, *libusb Documentation*, Open Source Software Documentation.

ΠΑΡΑΡΤΗΜΑ Α : Κώδικας

1. Adc

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file    adc.c
  * @brief   This file provides code for the configuration
  *          of the ADC instances.
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "adc.h"

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

ADC_HandleTypeDef hadc1;
ADC_HandleTypeDef hadc2;
ADC_HandleTypeDef hadc3;
DMA_HandleTypeDef hdma_adc1;
DMA_HandleTypeDef hdma_adc2;
DMA_HandleTypeDef hdma_adc3;

/* ADC1 init function */
void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_MultiModeTypeDef multimode = {0};
  ADC_ChannelConfTypeDef sConfig = {0};
```

```
/* USER CODE BEGIN ADC1_Init 1 */

/* USER CODE END ADC1_Init 1 */

/** Common config
*/
hadc1.Instance = ADC1;
hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.ContinuousConvMode = ENABLE;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DMA_CIRCULAR;
hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
hadc1.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
hadc1.Init.OversamplingMode = DISABLE;
hadc1.Init.Oversampling.Ratio = 1;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
  Error_Handler();
}
hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
hadc1.Init.Resolution = ADC_RESOLUTION_16B;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
  Error_Handler();
}

/** Configure the ADC multi-mode
*/
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
  Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_15;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
sConfig.OffsetSignedSaturation = DISABLE;
```

```
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}
/* ADC2 init function */
void MX_ADC2_Init(void)
{

  /* USER CODE BEGIN ADC2_Init 0 */

  /* USER CODE END ADC2_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC2_Init 1 */

  /* USER CODE END ADC2_Init 1 */

  /** Common config
  */
  hadc2.Instance = ADC2;
  hadc2.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc2.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc2.Init.LowPowerAutoWait = DISABLE;
  hadc2.Init.ContinuousConvMode = ENABLE;
  hadc2.Init.NbrOfConversion = 1;
  hadc2.Init.DiscontinuousConvMode = DISABLE;
  hadc2.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc2.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DMA_CIRCULAR;
  hadc2.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  hadc2.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
  hadc2.Init.OversamplingMode = DISABLE;
  hadc2.Init.Oversampling.Ratio = 1;
  if (HAL_ADC_Init(&hadc2) != HAL_OK)
  {
    Error_Handler();
  }
  hadc2.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc2.Init.Resolution = ADC_RESOLUTION_16B;
  if (HAL_ADC_Init(&hadc2) != HAL_OK)
  {
    Error_Handler();
  }
```

```
  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_10;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  sConfig.OffsetSignedSaturation = DISABLE;
  if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC2_Init 2 */

  /* USER CODE END ADC2_Init 2 */

}
/* ADC3 init function */
void MX_ADC3_Init(void)
{

  /* USER CODE BEGIN ADC3_Init 0 */

  /* USER CODE END ADC3_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC3_Init 1 */

  /* USER CODE END ADC3_Init 1 */

  /** Common config
  */
  hadc3.Instance = ADC3;
  hadc3.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc3.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc3.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc3.Init.LowPowerAutoWait = DISABLE;
  hadc3.Init.ContinuousConvMode = ENABLE;
  hadc3.Init.NbrOfConversion = 1;
  hadc3.Init.DiscontinuousConvMode = DISABLE;
  hadc3.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc3.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DMA_CIRCULAR;
  hadc3.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  hadc3.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
  hadc3.Init.OversamplingMode = DISABLE;
```

```
  hadc3.Init.Oversampling.Ratio = 1;
  if (HAL_ADC_Init(&hadc3) != HAL_OK)
  {
    Error_Handler();
  }
  hadc3.Init.Resolution = ADC_RESOLUTION_16B;
  if (HAL_ADC_Init(&hadc3) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_1;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  sConfig.OffsetSignedSaturation = DISABLE;
  if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC3_Init 2 */

  /* USER CODE END ADC3_Init 2 */

}

static uint32_t HAL_RCC_ADC12_CLK_ENABLED=0;

void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(adcHandle->Instance==ADC1)
  {
  /* USER CODE BEGIN ADC1_MspInit 0 */

  /* USER CODE END ADC1_MspInit 0 */
    /* ADC1 clock enable */
    HAL_RCC_ADC12_CLK_ENABLED++;
    if(HAL_RCC_ADC12_CLK_ENABLED==1){
      __HAL_RCC_ADC12_CLK_ENABLE();
    }

    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**ADC1 GPIO Configuration
```

```
  PA3     ------> ADC1_INP15
  */
  GPIO_InitStruct.Pin = GPIO_PIN_3;
  GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /* ADC1 DMA Init */
  /* ADC1 Init */
  hdma_adc1.Instance = DMA1_Stream0;
  hdma_adc1.Init.Request = DMA_REQUEST_ADC1;
  hdma_adc1.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_adc1.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_adc1.Init.MemInc = DMA_MINC_ENABLE;
  hdma_adc1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
  hdma_adc1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
  hdma_adc1.Init.Mode = DMA_CIRCULAR;
  hdma_adc1.Init.Priority = DMA_PRIORITY_LOW;
  hdma_adc1.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
  if (HAL_DMA_Init(&hdma_adc1) != HAL_OK)
  {
    Error_Handler();
  }

  __HAL_LINKDMA(adcHandle,DMA_Handle,hdma_adc1);

/* USER CODE BEGIN ADC1_MspInit 1 */

/* USER CODE END ADC1_MspInit 1 */
}
else if(adcHandle->Instance==ADC2)
{
/* USER CODE BEGIN ADC2_MspInit 0 */

/* USER CODE END ADC2_MspInit 0 */
  /* ADC2 clock enable */
  HAL_RCC_ADC12_CLK_ENABLED++;
  if(HAL_RCC_ADC12_CLK_ENABLED==1){
    __HAL_RCC_ADC12_CLK_ENABLE();
  }

  __HAL_RCC_GPIOC_CLK_ENABLE();
  /**ADC2 GPIO Configuration
  PC0     ------> ADC2_INP10
  */
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

```
  /* ADC2 DMA Init */
  /* ADC2 Init */
  hdma_adc2.Instance = DMA1_Stream1;
  hdma_adc2.Init.Request = DMA_REQUEST_ADC2;
  hdma_adc2.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_adc2.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_adc2.Init.MemInc = DMA_MINC_ENABLE;
  hdma_adc2.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
  hdma_adc2.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
  hdma_adc2.Init.Mode = DMA_CIRCULAR;
  hdma_adc2.Init.Priority = DMA_PRIORITY_LOW;
  hdma_adc2.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
  if (HAL_DMA_Init(&hdma_adc2) != HAL_OK)
  {
    Error_Handler();
  }

  __HAL_LINKDMA(adcHandle,DMA_Handle,hdma_adc2);

/* USER CODE BEGIN ADC2_MspInit 1 */

/* USER CODE END ADC2_MspInit 1 */
}
else if(adcHandle->Instance==ADC3)
{
/* USER CODE BEGIN ADC3_MspInit 0 */

/* USER CODE END ADC3_MspInit 0 */
  /* ADC3 clock enable */
  __HAL_RCC_ADC3_CLK_ENABLE();

  __HAL_RCC_GPIOC_CLK_ENABLE();
  /**ADC3 GPIO Configuration
  PC3_C     ------> ADC3_INP1
  */
  HAL_SYSCFG_AnalogSwitchConfig(SYSCFG_SWITCH_PC3, SYSCFG_SWITCH_PC3_OPEN);

  /* ADC3 DMA Init */
  /* ADC3 Init */
  hdma_adc3.Instance = BDMA_Channel0;
  hdma_adc3.Init.Request = BDMA_REQUEST_ADC3;
  hdma_adc3.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_adc3.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_adc3.Init.MemInc = DMA_MINC_ENABLE;
  hdma_adc3.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
  hdma_adc3.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
  hdma_adc3.Init.Mode = DMA_CIRCULAR;
  hdma_adc3.Init.Priority = DMA_PRIORITY_LOW;
```

```
    if (HAL_DMA_Init(&hdma_adc3) != HAL_OK)
    {
      Error_Handler();
    }

    __HAL_LINKDMA(adcHandle,DMA_Handle,hdma_adc3);

  /* USER CODE BEGIN ADC3_MspInit 1 */

  /* USER CODE END ADC3_MspInit 1 */
  }
}

void HAL_ADC_MspDeInit(ADC_HandleTypeDef* adcHandle)
{

  if(adcHandle->Instance==ADC1)
  {
  /* USER CODE BEGIN ADC1_MspDeInit 0 */

  /* USER CODE END ADC1_MspDeInit 0 */
    /* Peripheral clock disable */
    HAL_RCC_ADC12_CLK_ENABLED--;
    if(HAL_RCC_ADC12_CLK_ENABLED==0){
      __HAL_RCC_ADC12_CLK_DISABLE();
    }

    /**ADC1 GPIO Configuration
    PA3     ------> ADC1_INP15
    */
    HAL_GPIO_DeInit(GPIOA, GPIO_PIN_3);

    /* ADC1 DMA DeInit */
    HAL_DMA_DeInit(adcHandle->DMA_Handle);
  /* USER CODE BEGIN ADC1_MspDeInit 1 */

  /* USER CODE END ADC1_MspDeInit 1 */
  }
  else if(adcHandle->Instance==ADC2)
  {
  /* USER CODE BEGIN ADC2_MspDeInit 0 */

  /* USER CODE END ADC2_MspDeInit 0 */
    /* Peripheral clock disable */
    HAL_RCC_ADC12_CLK_ENABLED--;
    if(HAL_RCC_ADC12_CLK_ENABLED==0){
      __HAL_RCC_ADC12_CLK_DISABLE();
    }
```

```
    /**ADC2 GPIO Configuration
    PC0     ------> ADC2_INP10
    */
    HAL_GPIO_DeInit(GPIOC, GPIO_PIN_0);

    /* ADC2 DMA DeInit */
    HAL_DMA_DeInit(adcHandle->DMA_Handle);
  /* USER CODE BEGIN ADC2_MspDeInit 1 */

  /* USER CODE END ADC2_MspDeInit 1 */
  }
  else if(adcHandle->Instance==ADC3)
  {
  /* USER CODE BEGIN ADC3_MspDeInit 0 */

  /* USER CODE END ADC3_MspDeInit 0 */
    /* Peripheral clock disable */
    __HAL_RCC_ADC3_CLK_DISABLE();

    /* ADC3 DMA DeInit */
    HAL_DMA_DeInit(adcHandle->DMA_Handle);
  /* USER CODE BEGIN ADC3_MspDeInit 1 */

  /* USER CODE END ADC3_MspDeInit 1 */
  }
}

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */
```

2. Dma

bdma.c

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file    bdma.c
  * @brief   This file provides code for the configuration
  *          of all the requested memory to memory DMA transfers.
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
```

```
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "bdma.h"

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/*----------------------------------------------------------------------------*/
/* Configure DMA                                                              */
/*----------------------------------------------------------------------------*/

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */

/**
  * Enable DMA controller clock
  */
void MX_BDMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_BDMA_CLK_ENABLE();

  /* DMA interrupt init */
  /* BDMA_Channel0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(BDMA_Channel0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(BDMA_Channel0_IRQn);

}

/* USER CODE BEGIN 2 */

/* USER CODE END 2 */
```

dma.c

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file    dma.c
  * @brief   This file provides code for the configuration
  *          of all the requested memory to memory DMA transfers.
  ******************************************************************************
```

```
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "dma.h"

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/*----------------------------------------------------------------------------*/
/* Configure DMA                                                              */
/*----------------------------------------------------------------------------*/

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */

/**
  * Enable DMA controller clock
  */
void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Stream0_IRQn);
  /* DMA1_Stream1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Stream1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Stream1_IRQn);

}

/* USER CODE BEGIN 2 */
```

```
/* USER CODE END 2 */
```

3. Usb

usbd_custom_class.c

```
/**
  ******************************************************************************
  * @file    usbd_template.c
  * @author  MCD Application Team
  * @brief   This file provides the HID core functions.
  *
  * @verbatim
  *
  *          ===================================================================
  *                                CUSTOM_CLASS Class  Description
  *          ===================================================================
  *
  *
  *
  *
  *
  *
  *
  * @note     In HS mode and when the DMA is used, all variables and data
structures
  *           dealing with the DMA during the transaction process should be 32-bit
aligned.
  *
  *
  *  @endverbatim
  *
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2015 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under Ultimate Liberty license
  * SLA0044, the "License"; You may not use this file except in compliance with
  * the License. You may obtain a copy of the License at:
  *                       www.st.com/SLA0044
  *
  ******************************************************************************
  */

/* Includes ------------------------------------------------------------------*/
#include "usbd_custom_class.h"
#include "usbd_ctlreq.h"
```

```
/** @addtogroup STM32_USB_DEVICE_LIBRARY
  * @{
  */


/** @defgroup USBD_CUSTOM_CLASS
  * @brief usbd core module
  * @{
  */

/** @defgroup USBD_CUSTOM_CLASS_Private_TypesDefinitions
  * @{
  */
/**
  * @}
  */


/** @defgroup USBD_CUSTOM_CLASS_Private_Defines
  * @{
  */

/**
  * @}
  */


/** @defgroup USBD_CUSTOM_CLASS_Private_Macros
  * @{
  */

/**
  * @}
  */




/** @defgroup USBD_CUSTOM_CLASS_Private_FunctionPrototypes
  * @{
  */


static uint8_t USBD_CUSTOM_CLASS_Init(USBD_HandleTypeDef *pdev, uint8_t cfgidx);
static uint8_t USBD_CUSTOM_CLASS_DeInit(USBD_HandleTypeDef *pdev, uint8_t cfgidx);
static uint8_t USBD_CUSTOM_CLASS_Setup(USBD_HandleTypeDef *pdev,
USBD_SetupReqTypedef *req);
static uint8_t USBD_CUSTOM_CLASS_DataIn(USBD_HandleTypeDef *pdev, uint8_t epnum);
```

```
static uint8_t USBD_CUSTOM_CLASS_DataOut(USBD_HandleTypeDef *pdev, uint8_t epnum);
static uint8_t USBD_CUSTOM_CLASS_EP0_RxReady(USBD_HandleTypeDef *pdev);
static uint8_t USBD_CUSTOM_CLASS_EP0_TxReady(USBD_HandleTypeDef *pdev);
static uint8_t USBD_CUSTOM_CLASS_SOF(USBD_HandleTypeDef *pdev);
static uint8_t USBD_CUSTOM_CLASS_IsoINIncomplete(USBD_HandleTypeDef *pdev, uint8_t
epnum);
static uint8_t USBD_CUSTOM_CLASS_IsoOutIncomplete(USBD_HandleTypeDef *pdev,
uint8_t epnum);

static uint8_t *USBD_CUSTOM_CLASS_GetCfgDesc(uint16_t *length);
static uint8_t *USBD_CUSTOM_CLASS_GetDeviceQualifierDesc(uint16_t *length);
/**
  * @}
  */

/** @defgroup USBD_CUSTOM_CLASS_Private_Variables
  * @{
  */

static uint8_t data_out_buffer[CUSTOM_CLASS_MAX_PACKET_SIZE] = {};

/* this struct will store some internal class data */
USBD_CUSTOM_CLASS_DataTypeDef custom_class_data =
{
  data_out_buffer,
  NULL,
  CUSTOM_CLASS_MAX_PACKET_SIZE,
  CUSTOM_CLASS_MAX_PACKET_SIZE,
  0,
  0,
};


USBD_ClassTypeDef USBD_CUSTOM_CLASS_ClassDriver =
{
  USBD_CUSTOM_CLASS_Init,
  USBD_CUSTOM_CLASS_DeInit,
  USBD_CUSTOM_CLASS_Setup,            /* not used in the current class
implementation */
  USBD_CUSTOM_CLASS_EP0_TxReady,      /* not used in the current class
implementation */
  USBD_CUSTOM_CLASS_EP0_RxReady,      /* not used in the current class
implementation */
  USBD_CUSTOM_CLASS_DataIn,
  USBD_CUSTOM_CLASS_DataOut,
  USBD_CUSTOM_CLASS_SOF,              /* not used in the current class
implementation */
  USBD_CUSTOM_CLASS_IsoINIncomplete,  /* not used in the current class
implementation */
```

```
  USBD_CUSTOM_CLASS_IsoOutIncomplete, /* not used in the current class
implementation */
  USBD_CUSTOM_CLASS_GetCfgDesc, // get config desc for HS
  USBD_CUSTOM_CLASS_GetCfgDesc, // get config desc for FS
  USBD_CUSTOM_CLASS_GetCfgDesc, // get config desc for otherspeed
  USBD_CUSTOM_CLASS_GetDeviceQualifierDesc,
};



#if defined ( __ICCARM__ ) /*!< IAR Compiler */
#pragma data_alignment=4
#endif
/* USB CUSTOM_CLASS device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t
USBD_CUSTOM_CLASS_CfgDesc[USB_CUSTOM_CLASS_CONFIG_DESC_SIZ] __ALIGN_END =
{
  /* Configuration Descriptor */
  USB_LEN_CFG_DESC,              /* bLength: Configuation Descriptor size */
  USB_DESC_TYPE_CONFIGURATION,   /* bDescriptorType: Configuration */
  USB_CUSTOM_CLASS_CONFIG_DESC_SIZ,  /* wTotalLength: Bytes returned */
  0x00,
  0x01,                          /*bNumInterfaces: 1 interface*/
  0x01,                          /*bConfigurationValue: Configuration value*/
  0x00,                          /*iConfiguration */
  0xC0,                          /*bmAttributes: bus powered and Supports Remote
Wakeup */
  0x32,                          /*MaxPower 100 mA: this current is used for
detecting Vbus*/

  /*---------------------------------------------------------------------------*/

  /* Interface Descriptor */
  USB_LEN_IF_DESC,          /* bLength */
  USB_DESC_TYPE_INTERFACE,  /* bDescriptorType: */
  0x00,                     /* bInterfaceNumber */
  0x00,                     /* bAlternateSetting */
  0x02,                     /* bNumEndpoints */
  0xFF,                     /* bInterfaceClass: 0xFF=vendor specific, can be set
to 0x0A (CDC data) in this implementation*/
  0x00,                     /* bInterfaceSubClass */
  0x00,                     /* bInterfaceProtocol */
  0x00,                     /* iInterface */

  /*---------------------------------------------------------------------------*/

  /* Endpoint Descriptor */
  USB_LEN_EP_DESC,                      /* bLength */
  USB_DESC_TYPE_ENDPOINT,               /* bDescriptorType */
```

```
  CUSTOM_CLASS_EP1_OUT_ADDR,             /* bEndpointAddress */
  USBD_EP_TYPE_BULK,                     /* bmAttributes */
  LOBYTE(CUSTOM_CLASS_MAX_PACKET_SIZE), /* wMaxPacketSize */
  HIBYTE(CUSTOM_CLASS_MAX_PACKET_SIZE),
  0x00,                                  /* bInterval, ignore for bulk transfert */

  /*---------------------------------------------------------------------------*/

  /* Endpoint Descriptor */
  USB_LEN_EP_DESC,                       /* bLength */
  USB_DESC_TYPE_ENDPOINT,                /* bDescriptorType */
  CUSTOM_CLASS_EP1_IN_ADDR,              /* bEndpointAddress */
  USBD_EP_TYPE_BULK,                                /* bmAttributes */
  LOBYTE(CUSTOM_CLASS_MAX_PACKET_SIZE), /* wMaxPacketSize */
  HIBYTE(CUSTOM_CLASS_MAX_PACKET_SIZE),
  0x00                                   /* bInterval, ignore for bulk transfert */

};

#if defined ( __ICCARM__ ) /*!< IAR Compiler */
#pragma data_alignment=4
#endif
/* USB Standard Device Descriptor */
__ALIGN_BEGIN static uint8_t
USBD_CUSTOM_CLASS_DeviceQualifierDesc[USB_LEN_DEV_QUALIFIER_DESC] __ALIGN_END =
{
  USB_LEN_DEV_QUALIFIER_DESC,
  USB_DESC_TYPE_DEVICE_QUALIFIER,
  0x00,
  0x02,
  0x00,
  0x00,
  0x00,
  0x40,
  0x01,
  0x00,
};

/**
  * @}
  */

/** @defgroup USBD_CUSTOM_CLASS_Private_Functions
  * @{
  */

/**
  * @brief  USBD_CUSTOM_CLASS_Init
  *         Initialize the CUSTOM_CLASS interface
```

```
  * @param  pdev: device instance
  * @param  cfgidx: Configuration index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_Init(USBD_HandleTypeDef *pdev, uint8_t cfgidx)
{
  /* Open EP1 IN */
  USBD_LL_OpenEP(pdev, CUSTOM_CLASS_EP1_IN_ADDR, USBD_EP_TYPE_BULK,
CUSTOM_CLASS_MAX_PACKET_SIZE);
  pdev->ep_in[CUSTOM_CLASS_EP1_IN_ADDR & 0xFU].is_used = 1U;

  /* Open EP1 OUT */
  USBD_LL_OpenEP(pdev, CUSTOM_CLASS_EP1_OUT_ADDR, USBD_EP_TYPE_BULK,
CUSTOM_CLASS_MAX_PACKET_SIZE);
  pdev->ep_out[CUSTOM_CLASS_EP1_OUT_ADDR & 0xFU].is_used = 1U;

  /* Call Interface Init  */
  ((USBD_CUSTOM_CLASS_ItfTypeDef *)pdev->pUserData[pdev->classId])->Init();

  // register custom class data
  pdev->pClassData = (void *)&custom_class_data;

  /* Prepare Out endpoint to receive next packet */
  USBD_LL_PrepareReceive(pdev, CUSTOM_CLASS_EP1_OUT_ADDR,
custom_class_data.RxBuffer, custom_class_data.RxLength);
  return (uint8_t)USBD_OK;
}

/**
  * @brief  USBD_CUSTOM_CLASS_Init
  *         DeInitialize the CUSTOM_CLASS layer
  * @param  pdev: device instance
  * @param  cfgidx: Configuration index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_DeInit(USBD_HandleTypeDef *pdev, uint8_t cfgidx)
{
  /* Close EP1 IN */
  USBD_LL_CloseEP(pdev, CUSTOM_CLASS_EP1_IN_ADDR);
  pdev->ep_in[CUSTOM_CLASS_EP1_IN_ADDR & 0xFU].is_used = 0U;

  /* Close EP1 OUT */
  USBD_LL_CloseEP(pdev, CUSTOM_CLASS_EP1_OUT_ADDR);
  pdev->ep_out[CUSTOM_CLASS_EP1_OUT_ADDR & 0xFU].is_used = 0U;

  /* Call Interface DeInit  */
  ((USBD_CUSTOM_CLASS_ItfTypeDef *)pdev->pUserData[pdev->classId])->DeInit();

  pdev->pClassData = NULL;
```

```
  return (uint8_t)USBD_OK;
}

/**
  * @brief  USBD_CUSTOM_CLASS_Setup
  *         Handle the CUSTOM_CLASS specific requests
  * @param  pdev: instance
  * @param  req: usb requests
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_Setup(USBD_HandleTypeDef *pdev,
                                       USBD_SetupReqTypedef *req)
{
  USBD_StatusTypeDef ret = USBD_OK;

  switch (req->bmRequest & USB_REQ_TYPE_MASK)
  {
  case USB_REQ_TYPE_CLASS :
    switch (req->bRequest)
    {
    default:
      USBD_CtlError(pdev, req);
      ret = USBD_FAIL;
      break;
    }
    break;

  case USB_REQ_TYPE_STANDARD:
    switch (req->bRequest)
    {
    default:
      USBD_CtlError(pdev, req);
      ret = USBD_FAIL;
      break;
    }
    break;

  default:
    USBD_CtlError(pdev, req);
    ret = USBD_FAIL;
    break;
  }

  return (uint8_t)ret;
}


/**
```

```
  * @brief  USBD_CUSTOM_CLASS_GetCfgDesc
  *         return configuration descriptor
  * @param  length : pointer data length
  * @retval pointer to descriptor buffer
  */
static uint8_t *USBD_CUSTOM_CLASS_GetCfgDesc(uint16_t *length)
{
  *length = (uint16_t)sizeof(USBD_CUSTOM_CLASS_CfgDesc);
  return USBD_CUSTOM_CLASS_CfgDesc;
}

/**
* @brief  DeviceQualifierDescriptor
*         return Device Qualifier descriptor
* @param  length : pointer data length
* @retval pointer to descriptor buffer
*/
uint8_t *USBD_CUSTOM_CLASS_GetDeviceQualifierDesc(uint16_t *length)
{
  *length = (uint16_t)sizeof(USBD_CUSTOM_CLASS_DeviceQualifierDesc);

  return USBD_CUSTOM_CLASS_DeviceQualifierDesc;
}


/**
  * @brief  USBD_CUSTOM_CLASS_DataIn
  *         handle data IN Stage
  * @param  pdev: device instance
  * @param  epnum: endpoint index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_DataIn(USBD_HandleTypeDef *pdev, uint8_t epnum)
{
  USBD_CUSTOM_CLASS_DataTypeDef *data;
  // retreive custom class data
  data = (USBD_CUSTOM_CLASS_DataTypeDef *)pdev->pClassData;

  // A bulk transfer is complete when the endpoint does on of the following:
  // - Has transferred exactly the amount of data expected
  // - Transfers a packet with a payload size less than wMaxPacketSize or
transfers a zero-length packet
  if ( (pdev->ep_in[epnum].total_length > 0) &&
    ((pdev->ep_in[epnum].total_length % CUSTOM_CLASS_MAX_PACKET_SIZE) == 0U) )
  {
    /* Update the packet total length */
    pdev->ep_in[epnum].total_length = 0;

    /* Send ZLP */
```

```
    USBD_LL_Transmit(pdev, epnum, NULL, 0);
  }
  else
  {
    /* transmit complete */
    data->TxState = 0U; // flag data transmit done

    /* call interface transmit cplt callback */
    ((USBD_CUSTOM_CLASS_ItfTypeDef *)pdev->pUserData[pdev->classId])-
>TransmitCpltCallback(pdev, data->TxBuffer, data->TxLength, epnum);
  }

  return (uint8_t)USBD_OK;
}
/**
  * @brief  USBD_CUSTOM_CLASS_DataOut
  *         handle data OUT Stage
  * @param  pdev: device instance
  * @param  epnum: endpoint index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_DataOut(USBD_HandleTypeDef *pdev, uint8_t epnum)
{
  // USBD_CUSTOM_CLASS_DataTypeDef *data;
  // retreive custom class data
  // data = (USBD_CUSTOM_CLASS_DataTypeDef *)pdev->pClassData;

  size_t const bytes_received = USBD_LL_GetRxDataSize(pdev, epnum);

  USBD_CUSTOM_CLASS_ReceivedPacket(pdev, bytes_received);

  return (uint8_t)USBD_OK;
}

/**
  * @brief  USBD_CUSTOM_CLASS_EP0_RxReady
  *         handle EP0 Rx Ready event
  * @param  pdev: device instance
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_EP0_RxReady(USBD_HandleTypeDef *pdev)
{

  return (uint8_t)USBD_OK;
}
/**
  * @brief  USBD_CUSTOM_CLASS_EP0_TxReady
  *         handle EP0 TRx Ready event
  * @param  pdev: device instance
```

```
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_EP0_TxReady(USBD_HandleTypeDef *pdev)
{

  return (uint8_t)USBD_OK;
}
/**
  * @brief  USBD_CUSTOM_CLASS_SOF
  *         handle SOF event
  * @param  pdev: device instance
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_SOF(USBD_HandleTypeDef *pdev)
{

  return (uint8_t)USBD_OK;
}
/**
  * @brief  USBD_CUSTOM_CLASS_IsoINIncomplete
  *         handle data ISO IN Incomplete event
  * @param  pdev: device instance
  * @param  epnum: endpoint index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_IsoINIncomplete(USBD_HandleTypeDef *pdev, uint8_t
epnum)
{

  return (uint8_t)USBD_OK;
}
/**
  * @brief  USBD_CUSTOM_CLASS_IsoOutIncomplete
  *         handle data ISO OUT Incomplete event
  * @param  pdev: device instance
  * @param  epnum: endpoint index
  * @retval status
  */
static uint8_t USBD_CUSTOM_CLASS_IsoOutIncomplete(USBD_HandleTypeDef *pdev,
uint8_t epnum)
{

  return (uint8_t)USBD_OK;
}

/**
  * @brief  USBD_CUSTOM_CLASS_TransmitPacket
  *         Transmit packet on IN endpoint
  * @param  pdev: device instance
```

```
  * @retval status
  */
uint8_t USBD_CUSTOM_CLASS_TransmitPacket(USBD_HandleTypeDef *pdev, uint8_t* buf,
uint16_t length)
{
  USBD_StatusTypeDef ret = USBD_BUSY;

  USBD_CUSTOM_CLASS_DataTypeDef *data;
  // retreive custom class data
  data = (USBD_CUSTOM_CLASS_DataTypeDef *)pdev->pClassData;

  if(data->TxState) // transfert in progress (busy)
  {
    return USBD_BUSY;
  }

  data->TxBuffer = buf;
  data->TxLength = length;

  /* Update the packet total length */
  pdev->ep_in[CUSTOM_CLASS_EP1_IN_ADDR & 0xFU].total_length = length;

  /* Transmit next packet */
  data->TxState = 1U;
  ret = USBD_LL_Transmit(pdev, CUSTOM_CLASS_EP1_IN_ADDR, data->TxBuffer, data-
>TxLength);
  if (ret != USBD_OK)
  {
      data->TxState = 0U;
  }

  return (uint8_t)ret;
}

/**
  * @brief  USBD_CUSTOM_CLASS_ReceivedPacket
  *         callback when a packet was received
  *         prepare OUT Endpoint for the next packet reception
  * @param  pdev: device instance
  * @retval status
  */
uint8_t USBD_CUSTOM_CLASS_ReceivedPacket(USBD_HandleTypeDef *pdev, size_t const
bytes_received)
{
  USBD_CUSTOM_CLASS_DataTypeDef *data;
  // retreive custom class data
  data = (USBD_CUSTOM_CLASS_DataTypeDef *)pdev->pClassData;

  /* Call interface callback here to copy rx buffer to application */
```

```
  ((USBD_CUSTOM_CLASS_ItfTypeDef *)pdev->pUserData[pdev->classId])-
>ReceiveCallback(pdev, data->RxBuffer, bytes_received);

  /* Prepare Out endpoint to receive next packet */
  USBD_LL_PrepareReceive(pdev, CUSTOM_CLASS_EP1_OUT_ADDR, data->RxBuffer, data-
>RxLength);

  return (uint8_t)USBD_OK;
}

uint8_t USBD_CUSTOM_CLASS_RegisterInterface(USBD_HandleTypeDef *pdev,
USBD_CUSTOM_CLASS_ItfTypeDef *intf)
{
  if (intf == NULL)
  {
    return (uint8_t)USBD_FAIL;
  }

  /* refister the interface */
  pdev->pUserData[pdev->classId] = intf;

  return (uint8_t)USBD_OK;
}

/**
  * @}
  */


/**
  * @}
  */


/**
  * @}
  */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

usbd_custom_class_if.c

```
#include "usbd_custom_class_if.h"

static uint8_t Custom_Class_Init(void);
static uint8_t Custom_Class_DeInit(void);
//static uint8_t Custom_Class_Control(uint8_t cmd, uint8_t *pbuf, uint16_t
length);
```

```
static uint8_t Custom_Class_TransmitCplt_callback(USBD_HandleTypeDef* pDev,
uint8_t *pbuf, uint32_t len, uint8_t epnum);
static uint8_t Custom_Class_Receive_callback(USBD_HandleTypeDef* pDev, uint8_t
*pbuf, uint32_t len);

/* function for sending data */
uint8_t custom_class_transmit_data(USBD_HandleTypeDef* pDev, uint8_t *pbuf,
uint16_t len)
{
  return USBD_CUSTOM_CLASS_TransmitPacket(pDev, pbuf, len);
}

USBD_CUSTOM_CLASS_ItfTypeDef USBD_CUSTOM_CLASS_interface =
{
  Custom_Class_Init,
  Custom_Class_DeInit,
  // Custom_Class_Control, /* not used for this custom class implementation */
  Custom_Class_TransmitCplt_callback,
  Custom_Class_Receive_callback
};


static uint8_t Custom_Class_Init(void)
{
  /*  add application-level initialization if needed */
  return (uint8_t)USBD_OK;
}


static uint8_t Custom_Class_DeInit(void)
{
  /*  add application-level deinitialization if needed */
  return (uint8_t)USBD_OK;
}


/*static uint8_t Custom_Class_Control(uint8_t cmd, uint8_t *pbuf, uint16_t length)
{
  return (uint8_t)USBD_OK;
}*/


/* called by custom_class implementation when a transmit is complete */
static uint8_t Custom_Class_TransmitCplt_callback(USBD_HandleTypeDef* pDev,
uint8_t *Buf, uint32_t len, uint8_t epnum)
{
  usb_busy = 0;
  return (uint8_t)USBD_OK;
}
```

```
/* called by custom_class implementation when data are received */
static uint8_t Custom_Class_Receive_callback(USBD_HandleTypeDef* pDev, uint8_t
*Buf, uint32_t len)
{
  // echoing received data (for testing)
  // custom_class_transmit_data(pDev, Buf, len);

  return (uint8_t)USBD_OK;
}
```

4. Main

main.c:

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "adc.h"
#include "bdma.h"
#include "dma.h"
#include "eth.h"
#include "memorymap.h"
#include "usart.h"
#include "usb_device.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "usbd_custom_class_if.h"

/* USER CODE END Includes */
```

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
typedef struct __attribute__((packed))
{
    uint16_t adc1;
    uint16_t adc2;
    uint16_t adc3;
} adc_sample_t;

typedef struct __attribute__((packed))
{
    uint16_t ch1;
    uint16_t ch2;
    uint16_t ch3;
} adc_usb_sample_t;

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define USB_BUF_SIZE (100U)
#define USB_PACKET_SAMPLES  128
#define CUSTOM_CLASS_EPIN_SIZE  64
#define HALF_SAMPLES (ADC_BUF_SIZE / 2)

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */
ALIGN_32BYTES (uint16_t adc1_buf[ADC_BUF_SIZE]);
ALIGN_32BYTES (uint16_t adc2_buf[ADC_BUF_SIZE]);
__attribute__((section(".RAM_D3"))) uint16_t adc3_buf[ADC_BUF_SIZE];

volatile uint8_t adc1_half_ready = 0;
volatile uint8_t adc2_half_ready = 0;
volatile uint8_t adc3_half_ready = 0;

volatile uint8_t adc1_full_ready = 0;
volatile uint8_t adc2_full_ready = 0;
volatile uint8_t adc3_full_ready = 0;

volatile uint8_t half_ready = 0;
```

```
volatile uint8_t full_ready = 0;

volatile uint8_t usb_busy = 0;

static adc_usb_sample_t usb_tx_half[HALF_SAMPLES];
static adc_usb_sample_t usb_tx_full[HALF_SAMPLES];

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
void PeriphCommonClock_Config(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */
  memset(adc3_buf, 0, sizeof(adc3_buf)); // TODO Fix in linker script
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  HAL_PWREx_EnableUSBVoltageDetector();
  __HAL_RCC_HSI48_ENABLE();
  while (!__HAL_RCC_GET_FLAG(RCC_FLAG_HSI48RDY));
  MX_USB_DEVICE_Init();

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();
```

```
/* Configure the peripherals common clocks */
PeriphCommonClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_BDMA_Init();
MX_ETH_Init();
MX_USART3_UART_Init();
MX_ADC1_Init();
MX_ADC3_Init();
MX_ADC2_Init();
/* USER CODE BEGIN 2 */

HAL_ADC_Start_DMA(&hadc1, (uint32_t*) adc1_buf, ADC_BUF_SIZE);
HAL_ADC_Start_DMA(&hadc2, (uint32_t*) adc2_buf, ADC_BUF_SIZE);
HAL_ADC_Start_DMA(&hadc3, (uint32_t*) adc3_buf, ADC_BUF_SIZE);

HAL_Delay(100);

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
  if (!usb_busy)
  {
      if (half_ready)
      {
          half_ready = 0;

          for (uint16_t i = 0; i < HALF_SAMPLES; i++)
          {
              usb_tx_half[i].ch1 = adc1_buf[i];
              usb_tx_half[i].ch2 = adc2_buf[i];
              usb_tx_half[i].ch3 = adc3_buf[i];
          }

          usb_busy = 1;

          custom_class_transmit_data(
```

```
                &hUsbDeviceFS,
                (uint8_t*)usb_tx_half,
                sizeof(usb_tx_half)
            );
        }
        else if (full_ready)
        {
            full_ready = 0;

            for (uint16_t i = 0; i < HALF_SAMPLES; i++)
            {
                uint16_t idx = i + HALF_SAMPLES;

                usb_tx_full[i].ch1 = adc1_buf[idx];
                usb_tx_full[i].ch2 = adc2_buf[idx];
                usb_tx_full[i].ch3 = adc3_buf[idx];
            }

            usb_busy = 1;

            custom_class_transmit_data(
                &hUsbDeviceFS,
                (uint8_t*)usb_tx_full,
                sizeof(usb_tx_full)
            );
        }
    }
  }
    /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Supply configuration update enable
  */
  HAL_PWREx_ConfigSupply(PWR_LDO_SUPPLY);

  /** Configure the main internal regulator output voltage
  */
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  while(!__HAL_PWR_GET_FLAG(PWR_FLAG_VOSRDY)) {}
```

```
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 1;
  RCC_OscInitStruct.PLL.PLLN = 24;
  RCC_OscInitStruct.PLL.PLLP = 2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
  RCC_OscInitStruct.PLL.PLLR = 2;
  RCC_OscInitStruct.PLL.PLLRGE = RCC_PLL1VCIRANGE_3;
  RCC_OscInitStruct.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;
  RCC_OscInitStruct.PLL.PLLFRACN = 0;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2
                              |RCC_CLOCKTYPE_D3PCLK1|RCC_CLOCKTYPE_D1PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.SYSCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB3CLKDivider = RCC_APB3_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_APB1_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_APB2_DIV1;
  RCC_ClkInitStruct.APB4CLKDivider = RCC_APB4_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief Peripherals Common Clock Configuration
  * @retval None
  */
void PeriphCommonClock_Config(void)
{
  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

  /** Initializes the peripherals clock
```

```
  */
  PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_ADC;
  PeriphClkInitStruct.PLL2.PLL2M = 4;
  PeriphClkInitStruct.PLL2.PLL2N = 128;
  PeriphClkInitStruct.PLL2.PLL2P = 16;
  PeriphClkInitStruct.PLL2.PLL2Q = 2;
  PeriphClkInitStruct.PLL2.PLL2R = 2;
  PeriphClkInitStruct.PLL2.PLL2RGE = RCC_PLL2VCIRANGE_1;
  PeriphClkInitStruct.PLL2.PLL2VCOSEL = RCC_PLL2VCOWIDE;
  PeriphClkInitStruct.PLL2.PLL2FRACN = 0;
  PeriphClkInitStruct.AdcClockSelection = RCC_ADCCLKSOURCE_PLL2;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
```

```
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```