ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΠΕΙΡΑΙΑ
ΤΜΗΜΑ ΑΥΤΟΜΑΤΙΣΜΟΥ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΕΛΕΓΧΟΣ ΔΙΕΥΘΥΝΣΗΣ ΣΕ HOVERCRAFT

ΣΠΟΥΔΑΣΤΗΣ : ΤΣΑΛΛΟΣ ΒΑΣΙΛΕΙΟΣ

ΑΜ : 19219

ΥΠΕΥΘΥΝΟΣ : ΧΑΜΗΛΟΘΩΡΗΣ ΓΕΩΡΓΙΟΣ

ΠΕΡΙΕΧΟΜΕΝΑ

Εισαγωγή

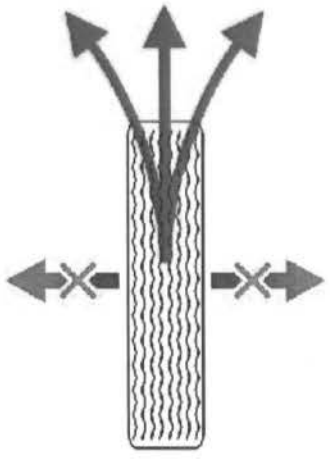
Το Hovercraft είναι ένα όχημα που χρησιμοποιεί ένα στρώμα αέρα υψηλής πίεσης, ανάμεσα στο όχημα και στην επιφάνεια κίνησης. Η πίεση αυτή εξισορροπεί το βάρος του και πρακτικά αιωρείται. Έτσι οι τριβές με την επιφάνεια κίνησης είναι σχεδόν μηδενικές. Μπορεί να κινείται σε όλες τις επιφάνειες, στεριά θάλασσα πάγο, αρκεί να μην υπάρχουν μεγάλες ανωμαλίες και η κλίση του εδάφους να είναι μικρή. Το στρώμα αέρα, δημιουργείται από μια εύκαμπτη φούσκα που φτάνει σχεδόν στην επιφάνεια του εδάφους. Έτσι το όχημα αιωρείται σε ύψος λίγων εκατοστών. Τα πρώτα λειτουργικά μοντέλα εμφανίστηκαν την δεκαετία του 50 αν και τα πρώτα σχέδια τοποθετούνται στις αρχές του 1700. Τα Hovercraft σήμερα χρησιμοποιούνται για μεταφορά επιβατών και οχημάτων, σαν οχήματα διάσωσης, αλλά και για στρατιωτικούς σκοπούς, συνήθως για άμεση μεταφορά βαρέως εξοπλισμού. Τέλος χρησιμοποιούνται για αναψυχή και ενίοτε και σε αγώνες.

Στην εργασία αυτή θα ασχοληθούμε με την κατασκευή ενός υπό κλίμακα μοντέλου, και στην δημιουργία ενός ψηφιακού συστήματος αυτομάτου ελέγχου προκειμένου να δείξουμε κατά πόσο μπορεί να γίνει έλεγχος της διεύθυνσης του οχήματος. Στόχος μας λοιπόν είναι περισσότερο ο πρακτικός έλεγχος σε μια κατασκευή και όχι η σε βάθος μαθητική ανάλυση και μοντελοποίηση του συστήματος, όπου και θα αναφερθούμε επιγραμματικά.

Το πρόβλημα ελέγχου

Το πρόβλημα ελέγχου της πορείας ενός hovercraft καθορίζεται από την ίδια την φύση του οχήματος, και ξεκινάει ουσιαστικά από την παντελή απουσία τριβών με την επιφάνεια κίνησης. Στα περισσότερα μέσα είναι ορισμένο από κατασκευής ότι η διεύθυνση της κίνησης συμπίπτει απόλυτα ή σε μεγάλο βαθμό με τον οριζόντιο άξονα συμμετρίας του εκάστοτε οχήματος, πχ. Αυτοκίνητα, αεροπλάνα, πλοία. Τρένα. Και κάθε κατασκευή υιοθετεί διαφορετικούς τρόπους για να εκτρέψει τον άξονα της, και κατά συνέπεια, να αλλάξει την πορεία του, καθώς ο άξονας συμμετρίας και η διεύθυνση της κίνησης πρέπει να συμπίπτουν.

Ας δούμε σαν παράδειγμα τα τροχοφόρα οχήματα και πιο συγκεκριμένα το αυτοκίνητο. Το αυτοκίνητο είναι κατ'αρχάς φτιαγμένο να κινείται σε μια διεύθυνση μπροστά ή πίσω παράλληλα με τον άξονα του. Η όλη μηχανική της κίνησης του βασίζεται στον τροχό. Ο τροχός έχει μηδενικές τριβές στην κίνηση μπρος πίσω, αλλά για να κινηθεί ένας τροχός πλάγια πρέπει να ασκηθεί μεγάλη δύναμη, μεγαλύτερη από την τριβή ολίσθησης μεταξύ του εδάφους και του ελαστικού, κάτι που συμβαίνει μόνο σε ακραίες συνθήκες (σχ1). Επίσης ο τροχός μπορεί να αλλάξει διεύθυνση, να περιστραφεί, σχετικά εύκολα, ιδιαίτερα ενώ κινείται. Ένα τετράτροχο όχημα λοιπόν κινείται αρχικά ευθεία παράλληλα με την διεύθυνση των τεσσάρων τροχών του, και προκείμενου να αλλάξει την διεύθυνση του, στρέφει του δυο τροχούς, με τρόπο ώστε οι άξονες και των τεσσάρων τροχών να τέμνονται στο ίδιο σημείο (σχ2). Αυτό είναι προϋπόθεση για να μην υπάρχει καθόλου πλάγια μετακίνηση σε κανέναν τροχό και να ελαχιστοποιηθούν οι τριβές.

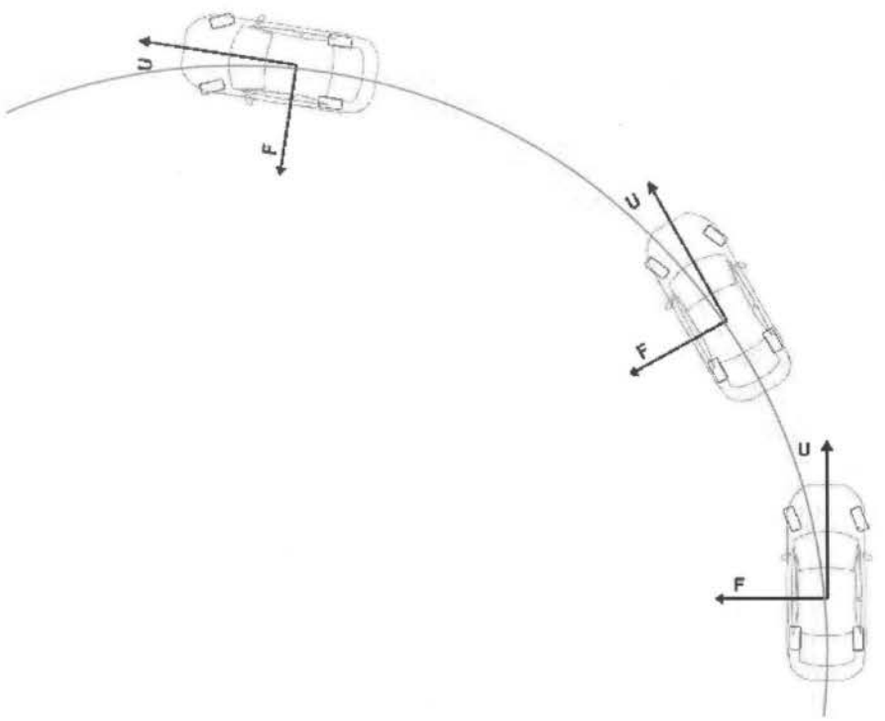


Σχ.1

Σχ.2

Στην πράξη αυτό δεν ισχύει απόλυτα σε όλα τα σύγχρονα τροχοφόρα (πχ. Πολυαξονικά φορτηγά με δυο σταθερούς, παράλληλους μεταξύ τους, άξονες) αλλά αποτελεί βασική αρχή. Το όχημα λοιπόν αφού στρέψει τους πρόσθιους τροχούς του θα ακολουθήσει μια κυκλική τροχιά με κέντρο το σημείο που τέμνονται οι τέσσερις τροχοί. Η κεντρομόλος δύναμη που απαιτείται για την διατήρηση της κυκλικής τροχιάς παρέχεται από τους τροχούς που αντιστέκονται σε κάθε πλάγια μετακίνηση τους. Η δύναμη αυτή είναι που ουσιαστικά αλλάζει την διεύθυνση της κίνησης του

οχήματος όταν αυτό στρίβει. Βλέπουμε λοιπόν ότι το τροχοφόρο υιοθετεί την στρέψη των δυο τροχών του για να αλλάξει την διεύθυνση του άξονα του, και η αλλαγή αυτή έχει ως επακόλουθο την αλλαγή της διεύθυνσης της ταχύτητας η οποία παραμένει ουσιαστικά παράλληλη στον άξονα του οχήματος ακολουθώντας εφαπτομενικά την κυκλική τροχιά (σχ3). και βασίζεται στην τριβή για να επιτύχει την αλλαγή της πορείας
του.



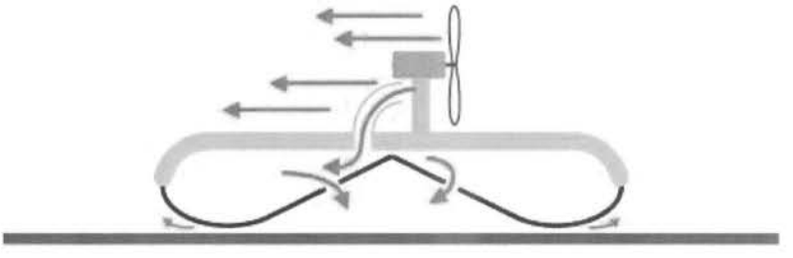
Σχ.3

Αντίστοιχοι μηχανισμοί ισχύουν για τα περισσότερα οχήματα: τα πλοία έχουν υδροδυναμικό σχήμα που τους επιτρέπει ευθεία κίνηση με πολύ λιγότερες τριβές από ότι χρειάζεται για κάθετη μετακίνηση, και χρησιμοποιούν το πηδάλιο και τις προπέλες για να εκτρέψουν συνήθως την πρύμνη και έτσι να αλλάξουν διεύθυνση. Τα αεροπλάνα χρησιμοποιούν το ουραίο πτερύγιο για να σταθεροποιεί την κίνηση στον αέρα παράλληλη με τον άξονα συμμετρίας τους και για να αλλάξουν την πορεία τους. Παρόλο που οι αποκλίσεις σε αυτή την περίπτωση είναι λίγο μεγαλύτερες από άλλα μέσα, μεγάλες αποκλίσεις είναι ικανές να προκαλέσουν απώλεια στήριξης και ελέγχου του αεροπλάνου. Μοναδική ίσως άλλη εξαίρεση πλέων του hovercraft αποτελεί το ελικόπτερο. Σε αντίθεση λοιπόν με τα περισσότερα μέσα ένα hovercraft δεν προϋποθέτει από κατασκευής ότι η διεύθυνση της ταχύτητας και ο άξονας συμμετρίας να είναι παράλληλοι. Εάν παραδείγματος χάρη ένα hovercraft έχει αναπτύξει μια γραμμική ταχύτητα, και αρχίσει να περιστρέφεται δεν υπάρχουν οι τριβές που θα εκτρέψουν την πορεία και ως εκ τούτου θα συνεχίσει να κινείται λίγο έως πολύ ευθεία ανεξάρτητα της διευθύνσεις του άξονα του. Και έτσι παρόλο που θα υπάρξει κάποια μεταβολή στη διεύθυνση της κίνησης, αν η περιστροφή συνεχισθεί το

όχημα μπορεί να μετακινείται προς μια κατεύθυνση και ο άξονας του να έχει την ακριβώς αντίθετη, και να εξακολουθεί να περιστρέφεται πλέον εκτός ελέγχου. Βασικός σκοπός αυτής της εργασίας είναι να ελεγχθεί αυτή η απόκλιση της διεύθυνσης της κίνησης σε σχέση με τον άξονα του οχήματος, προκειμένου να γίνει πιο εύκολος ο έλεγχος της πορείας ενός τέτοιου οχήματος.

Περιγραφή λειτουργίας hovercraft

Το hovercraft βασίζει την λειτουργία του στη φούσκα που βρίσκεται στο κάτω μέρος του οχήματος. Όταν αυτή η φούσκα γεμίσει με αέρα (σχ4), το βάρος εξισορροπείται από την επιπλέον πίεση αέρα κάτω από το όχημα και παύει να στηρίζεται στο έδαφος. Πλέων οι τριβές στην κίνηση είναι πολύ μικρές και το όχημα είναι ελεύθερο να κινηθεί σε κάθε κατεύθυνση. Η απαιτούμενη πίεση αέρα εξασφαλίζεται είτε δεσμεύοντας ένα μέρος της ροής του αέρα πίσω από την κεντρική έλικα, είτε με ξεχωριστό σύστημα ελίκων επιφορτισμένο με αυτό το έργο.

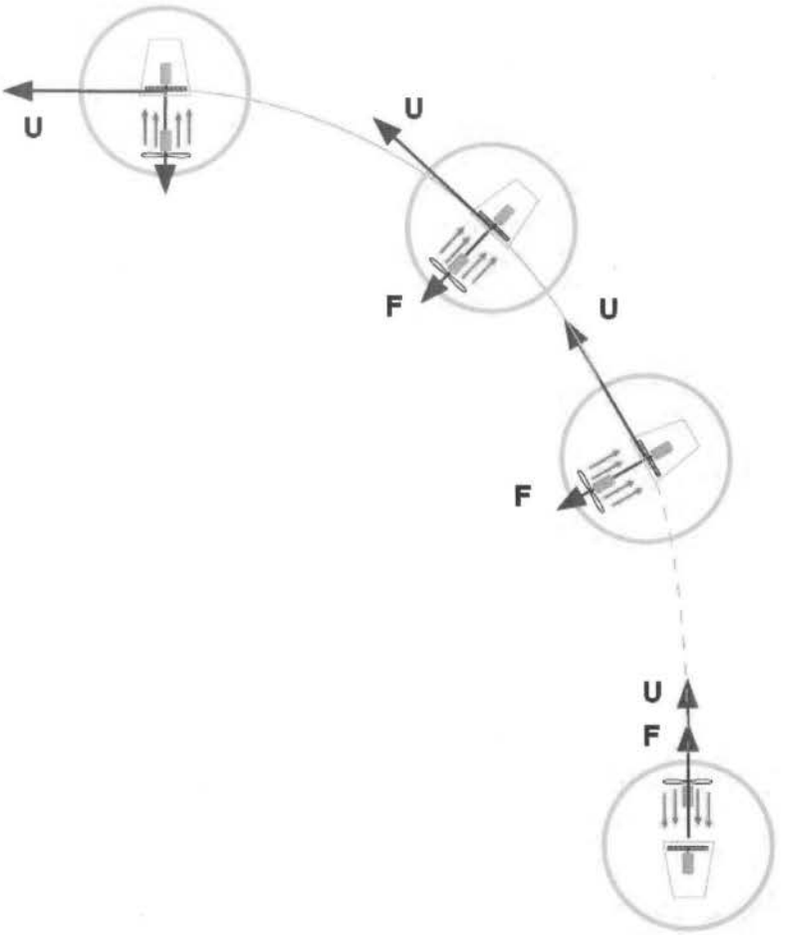


Σχ.4

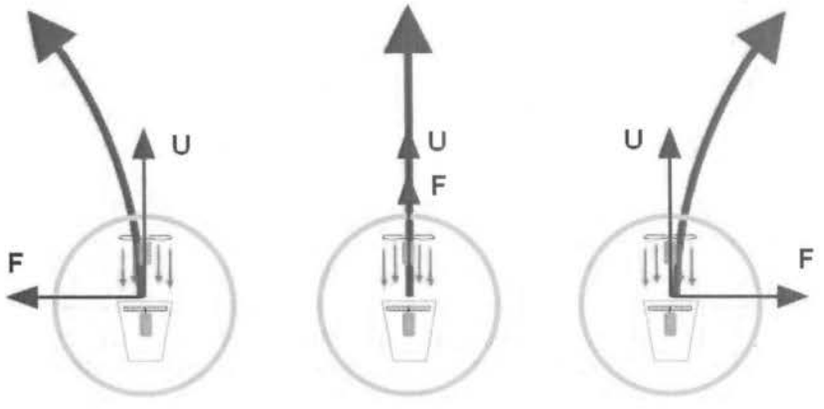
Το σύστημα της άνωσης λοιπόν απλά απαλείφει τις τριβές με το έδαφος αλλά δεν κατευθύνει το όχημα, δεν προκαλεί κάποια κίνηση. Η Δύναμη που δημιουργεί είναι στον κάθετο άξονα αντίθετη με το βάρος του. Κάθε δύναμη στο οριζόντιο επίπεδο παρέχεται από το σύστημα της προώθησης. Το σύστημα αυτό αποτελείται από μια η περισσότερες έλικες και τους αντίστοιχους εκτραπείς της ροής, πηδάλια. Τα πηδάλια απλώς εκτρέπουν την ροή αέρα δεξιά ή αριστερά και δεν δημιουργούν κάποια επιπλέον δύναμη απλά αλλάζουν την διεύθυνση της ώθησης που παράγει η εκάστοτε έλικα. Η χρησιμότητα τους είναι στο να ελέγξουν την περιστροφή του οχήματος και όχι την κατεύθυνση του.

Γίνεται λοιπόν κατανοητό ότι απουσία τριβών και οποιοδήποτε αλλού συστήματος ικανού να εκτρέψει την πορεία του οχήματος, αυτό πρέπει να γίνει βασιζόμενο στη ώθηση της κεντρικής έλικας είτε αυτή είναι μια είτε περισσότερες. Η ίδια δύναμη που χρησιμοποιείται για να επιταχύνει το όχημα σε ευθεία είναι η ίδια που παίζει το ρόλο κεντρομόλου επιτάχυνσης ώστε να κρατήσει το όχημα σε κυκλική τροχιά. Για να επιτευχθεί αυτό το όχημα πρέπει να περιστραφεί μέχρις ότου ο άξονας του οχήματος γίνει κάθετος στην διεύθυνση της κίνησης και να παραμείνει κάθετος καθ΄όλη την διάρκεια της κυκλικής πορείας (σχ5).Πρέπει λοιπόν να είμαστε σε θέση

να ελέγξουμε την γωνία που σχηματίζει ο άξονας του οχήματος σε σχέση με την διεύθυνση της κίνησης και όχι σε σχέση με κάποιο εξωτερικό σύστημα αναφοράς. Η γωνία αυτή πρέπει να είναι 0° όταν το όχημα επιταχύνει και ±90° όταν θέλουμε να στρίψουμε δεξιά-αριστερά αντίστοιχα (σχ6).



Σχ.5



Σχ.6

Περιγραφή κατασκευής

Ο αισθητήρας

Ένα από τα πιο βασικά κομμάτια της κατασκευής είναι η επιλογή του αισθητήρα που θα κάνει δυνατό τον απαιτούμενο έλεγχο.

Αυτό που θέλουμε να μετρήσουμε είναι η γωνία που σχηματίζει ο άξονας του hovercraft σε σχέση με την διεύθυνση της κίνησης.
Παραθέτω παρακάτω κάποιες από τις επιλογές που μελετήθηκαν.

Ψηφιακό επιταχυνσιόμετρο. Υπάρχουν στην αγορά ψηφιακά επιταχυνσιόμετρα τα οποία δίνουν σαν έξοδο την επιτάχυνση που δέχεται το ολοκληρωμένο σε δυο άξονες όπως π.χ. το LIS2L02AS4. Η χρησιμότητα του στην συγκεκριμένη κατασκευή θα ήταν αν μπορούσαμε μετρώντας την επιτάχυνση σε δυο άξονες και ολοκληρώνοντας την στο χρόνο να μπορέσουμε να εξάγουμε την ταχύτητα στους δυο άξονες X και Y και στην συνέχεια από τις δυο ταχύτητες να πάρουμε την γωνία της διεύθυνσης της ταχύτητας. Αυτό στην πραγματικότητα είναι αδύνατο διότι το ολοκληρωμένο θα περιστρέφεται μαζί με την κατασκευή και την περιστροφή δεν μπορούμε να την λάβουμε υπ όψιν.

Ψηφιακό γυροσκόπιο (π.χ ADXRS300): Μπορεί να μας δώσει μια σταθερή γωνία αναφοράς* και συνδυαζόμενο με ένα επιταχυνσιόμετρο να μπορέσουμε να εξάγουμε την γωνία που μας ενδιαφέρει. Παρ όλα αυτά ο μετασχηματισμός θα είναι εξαιρετικά πολύπλοκος πράγμα που απαιτεί μεγάλη επεξεργαστική ισχύ, και μειώνει την ακρίβεια. Επιπλέον αυτά τα ολοκληρωμένα έχουν σημαντικό κόστος.

Ψηφιακή πυξίδα: Πιο οικονομική εναλλακτική από την προηγούμενη λύση καθώς αυτό που μας δίνει είναι πάλι μια γωνία σε σχέση με μια εξωτερική αναφορά

---

* η γωνία αυτή σχηματίζεται ανάμεσα στην διεύθυνση του οχήματος και μια εξωτερική διεύθυνση αναφοράς ουσιαστικά μας λύνει το μισό μόνο πρόβλημα για αυτό είναι απαραίτητοι οι μετασχηματισμοί.

αλλά δε λύνει κάποιο από τα υπόλοιπα μειονεκτήματα που την κάνουν μη αποδεκτή ως λύση.

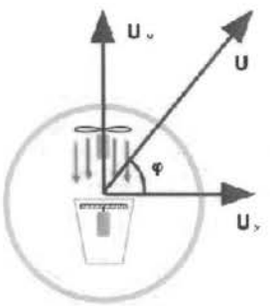
GPS: Αυτή η λύση συγκεντρώνει αρκετά πλεονεκτήματα, μπορεί να μας δώσει κατευθείαν την κατεύθυνση της κίνησης και δίνει την δυνατότητα ο προγραμματισμός να γίνει κατευθείαν στην συσκευή που φιλοξενεί τον αισθητήρα και γίνεται έτσι διαθέσιμη μεγάλη επεξεργαστική ισχύς. Το κόστος δεν είναι απαγορευτικό. Μοναδικό μειονέκτημα της λύσης αυτής είναι ότι σαν αισθητήρας έχει μια καθυστέρηση στην ανανέωση της εξόδου λίγων δευτερολέπτων. Αυτό θα μπορούσε να μην είναι πρόβλημα σε ένα πιο μεγάλο και αργό μοντέλο αλλά στο μέγεθος αυτής της εργασίας είναι απαγορευτικό.

Η λύση στο πρόβλημα του αισθητήρα που επιλέχθηκε για την συγκεκριμένη εφαρμογή είναι ένα ποντίκι υπολογιστή, μια συσκευή κατάδειξης δηλαδή. Η λύση αυτή έχει το πλεονέκτημα ότι μπορεί να μετρήσει κατευθείαν την μεταβλητή που θέλουμε. Στερεώνοντας το ποντίκι πάνω στην κατασκευή, ώστε να διαβάζει το έδαφος, ο άξονας του συμπίπτει με τον άξονα συμμετρίας του οχήματος, και μας δίνει σαν έξοδο την μετακίνηση σε δυο άξονες X και Y σε σταθερά χρονικά διαστήματα. Έχουμε δηλαδή την ταχύτητα μετακίνησης του οχήματος σε σχέση με το έδαφος σε δυο άξονες. Η ουσιαστική διαφορά με άλλες λύσεις είναι ότι δεν επηρεάζεται από την περιστροφή του οχήματος καθώς δεν μετράει την ταχύτητα σε σχέση με κάποιο σταθερό εξωτερικό σύστημα αναφοράς αλλά σε σχέση με το ίδιο το έδαφος ανεξάρτητα από την γωνία της διεύθυνσης του οχήματος. Το μόνο που χρειάζεται να κάνουμε εμείς είναι, από τις δυο συνιστώσες της ταχύτητας να εξάγουμε την γωνία της ταχύτητας (σε σχέση πάντα με τον άξονα του οχήματος) κάνοντας έναν απλό μετασχηματισμό (σχ7).

Αρχικά μελετήθηκε αν θα ήταν δυνατό να ελεγχθεί κατευθείαν ένας αισθητήρας ποντικιού. Σε ένα μηχανικό ποντίκι είναι αρκετά εύκολο να επέμβεις συνδέοντας κατευθείαν την οπτικές του διόδους με εισόδους ενός μικροεπεξεργαστή αλλά όπως αποδείχτηκε ένα μηχανικό ποντίκι δεν θα μπορούσε να δουλέψει σε μια τέτοια εφαρμογή διότι θέλει μεγάλη ακρίβεια στην στερέωση, χρειάζεται κάποιο έστω μικρό βάρος να το πιέζει στην επιφάνεια που διαβάζει για να λειτουργήσει, και επηρεάζεται περισσότερο από τυχών ανωμαλίες του εδάφους και ξένα υλικά που προσκολλώνται στην σφαίρα του.

Εναλλακτικά θα μπορούσε να ελεηθεί ένας αισθητήρας οπτικού ποντικιού (π.χ. το Agilent ADNS 2051) η επιλογή αυτή θα μείωνε πολύ το μέγεθος του αισθητήρα αλλά στην ουσία, και πολύ περίπλοκη είναι, και δεν προσφέρει κάποιο ουσιαστικό πλεονέκτημα καθώς στην αγορά υπάρχουν λύσεις σε αρκετά μικρό μέγεθος και η υλοποίηση τους προσφέρει σίγουρα αρκετή ακρίβεια. Η επιλογή ενός έτυμου προϊόντος απαιτεί την επικοινωνία με την συσκευή μέσο κάποιου πρωτοκόλλου PS2 ή USB. Πρέπει δηλαδή να συνδέσουμε έναν μικροεπεξεργαστή με το ποντίκι και να γράψουμε το πρόγραμμα οδήγησης (Driver) που θα κάνει την επικοινωνία μεταξύ των δυο συσκευών δυνατή. Αυτό μελετάται σε επόμενα κεφάλαια.

Ανακεφαλαιώνοντας λοιπόν τα πλεονεκτήματα της λύσης που επιλέξαμε είναι άμεση μέτρηση της ελεγχόμενης μεταβλητής, το χαμηλό κόστος και η μεγάλη γκάμα επιλογών στην αγορά. Στα μειονεκτήματα η ανάγκη δημιουργίας προγράμματος οδήγησης (Driver), και η αδυναμία λειτουργίας πανό από νερό. Ο περιορισμός στην επιφάνεια δεν αποτελεί πρόβλημα διότι αυτό δεν αποτελεί στόχο αυτής της εργασίας αλλά περισσότερο να δήξει ότι ο έλεγχος μπορεί να γίνει έστω και με κάποιες προϋποθέσεις.



Σχ.7 Σχ.8

Το σύστημα της άνωσης

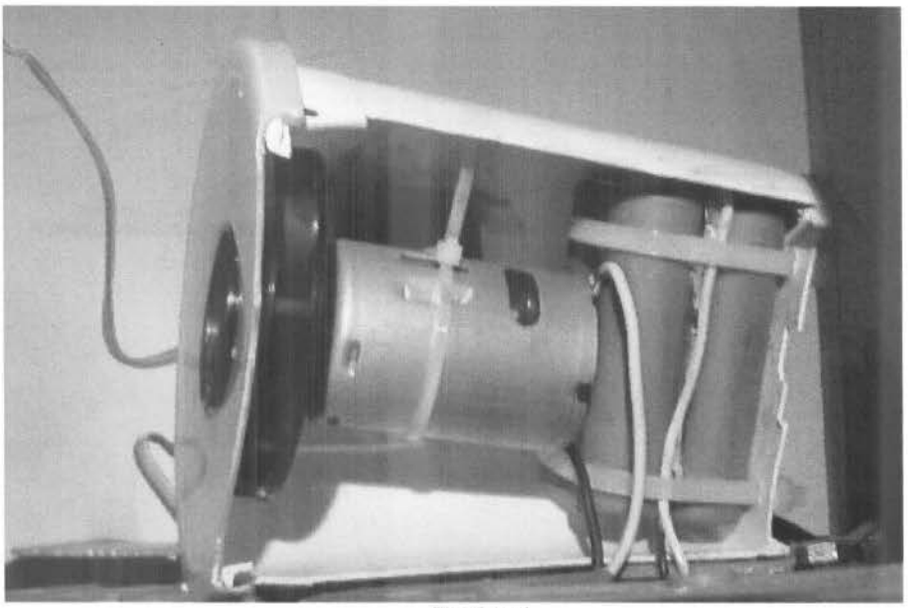
Βασικό κομμάτι της κατασκευής και το πρώτο που πρέπει να υλοποιηθεί είναι το σύστημα της άνωσης.

Καθώς το σύστημα αυτό περιλαμβάνει την φούσκα η οποία είναι η πλατφόρμα πάνω στην οποία θα προστεθούν μετά όλα τα υπόλοιπα κομμάτια, έπρεπε να καταλήξουμε στο πως θα υλοποιηθεί πριν από κάποιο άλλο κομμάτι της κατασκευής.

Αρχικά έγιναν πειραματισμοί με ένα ηλεκτρικό μοτέρ και μια φτερωτή που είχαν αφαιρεθεί από ένα επαναφορτιζόμενο ηλεκτρικό σκουπάκι. Οι λόγοι που οδήγησαν σε αυτή την επιλογή είναι ότι το hovercraft έχει γενικά μεγάλες δυνατότητες μεταφοράς βάρους όποτε το βάρος του κινητήρα δεν είναι ιδιαίτερα σημαντικό. Ακόμα η φτερωτή αυτού του τύπου είναι σχεδιασμένη να λειτουργεί με τον συγκεκριμένο κινητήρα, και έχει την δυνατότητα να παράξει την απαιτούμενη πίεση μέσα στην φούσκα, που είναι απαραίτητη για να σηκώσει το βάρος της κατασκευής. Η ισχύς του κινητήρα είναι στα 30W και τροφοδοτείται από μπαταρίες 4.8V. Στην πορεία της κατασκευής ο συγκεκριμένος κινητήρας συνδέθηκε με τάση μέχρι 8.4V, χωρίς να αντιμετωπίσει κάποιο πρόβλημα υπερθέρμανσης.

Σαν πρώτη προσπάθεια λοιπόν τοποθετήθηκε το σύστημα κινητήρα-φτερωτής σε ένα δοχείο με τον άξονα κάθετο στο έδαφος (σχ8). Το πρώτο πείραμα έδειξε ότι ο κινητήρας με την φτερωτή είναι όντως ικανός να σηκώσει αρκετό βάρος για όλη την κατασκευή αλλά όταν ήταν σε λειτουργία η κατασκευή αυτή περιστρεφόταν γύρω από τον εαυτό της. Ο λόγος που συμβαίνει αυτό είναι διότι καθώς ο αέρας εισέρχεται στην φτερωτή, δημιουργεί μια αντίσταση στην ροπή που ασκεί ο κινητήρας. Η αντίδραση της ροπής αυτής ασκείται στην κατασκευή η οποία είναι κατά τα αλλά μονωμένη από το περιβάλλον καθώς ίπταται και ως ετούτου περιστρέφεται. Μοναδική εναλλακτική για να λυθεί αυτό το πρόβλημα είναι τόσο ο κινητήρας της άνωσης όσο και κάθε άλλο ηλεκτρικό μοτέρ που τυχών χρησιμοποιηθεί να τοποθετηθεί έτσι ώστε ο άξονας του να είναι οριζόντιος με το έδαφος. Σαν πηγή τροφοδοσίας αρχικά χρησιμοποιήθηκε μια μπαταρία τεχνολογίας μολύβδου 6V, 3Ah κλειστού τύπου που κανονικά χρησιμοποιείται σε UPS. Το κριτήριο για την επιλογή

της μπαταρίας είναι οι μικρές διαστάσεις και συγκεκριμένα το μικρό ύψος που στη συγκεκριμένη περίπτωση περιορίζεται στα 33mm.



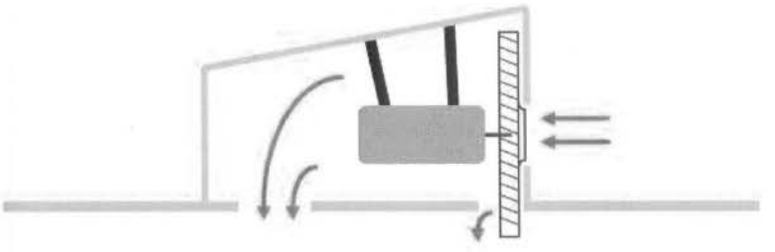
Εικόνα 1.

Καθώς λοιπόν είναι δεδομένο ότι ο άξονας του κινητήρα της άνωσης θα πρέπει να είναι οριζόντιος, η επιφάνεια από όπου εισέρχεται ο αέρας στην φτερωτή θα πρέπει να κάθετη στο έδαφος. Προκειμένου να υλοποιηθεί αυτό επιλέχθηκε ένα μικρό δοχείο το οποίο φιλοξενεί τον κινητήρα με την φτερωτή (εικ1) και το οποίο κόπηκε κάθετα στο πλάι ώστε να ενωθεί με την πλατφόρμα που θα φιλοξενήσει την φούσκα (σχ9). Ο κινητήρας στερεώθηκε απ'ευθείας στο δοχείο με μια πλαστική βάση ώστε οι ανοχές μεταξύ της φτερωτής και της αντίστοιχης τρύπας για την εισαγωγή του αέρα να είναι κατά το δυνατόν οι μικρότερες προκειμένου να διαφεύγει λιγότερη ποσότητα αέρα. Σαν πλατφόρμα στήριξης της φούσκας χρησιμοποιήθηκε ένας στρογγυλός δίσκος διαμέτρου 35εκ. Τέλος τα δυο μέρη ενώθηκαν μεταξύ τους. Στις δοκιμές η κατασκευή έδειξε ότι μπορεί να σηκώσει αρκετό βάρος και οι τριβές στην κίνηση με το έδαφος ήταν πολύ περιορισμένες.

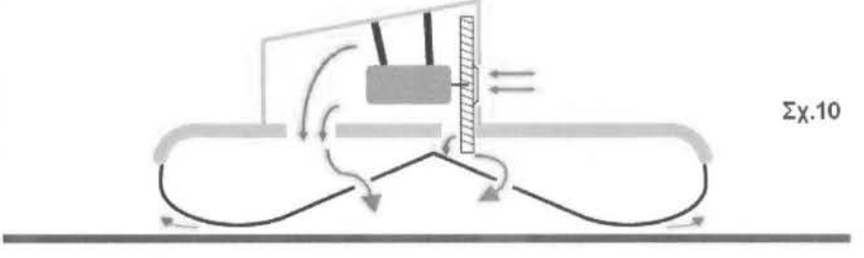
Οι μέχρι τώρα δοκιμές έχουν γίνει χωρίς την φούσκα. Αυτό είναι δυνατόν μόνο σε τελείως επίπεδη επιφάνεια. Για να μειωθούν περαιτέρω οι τριβές και να μπορεί το όχημα να περνάει μικρές ανωμαλίες του εδάφους, καθώς και για να είναι πιο ρεαλιστικό το μοντέλο θα έπρεπε να προστεθεί και αυτό το κομμάτι. Η φούσκα που κατασκευάσθηκε από ύφασμα που ενώθηκε με την περιφέρεια του δίσκου που χρησιμοποιούμε ως βάση. Τέλος απαιτείται να τραβήξουμε το κέντρο της φούσκας προς τα πάνω ώστε να δημιουργηθεί ο θύλακας αέρα με μεγαλύτερη πίεση και βεβαίως να ανοιχθούν οπές στο κάτω μέρος του υλικού της φούσκας ώστε να μπορεί ο αέρας να περνάει, το αποτέλεσμα φαίνεται στο σχήμα (σχ10). Το μέγεθος των οπών δεν είναι πολύ σημαντικό μπορούν να είναι αρκετά μεγάλες αρκεί να είναι μέσα στον θύλακα αέρα που θα δημιουργηθεί δηλαδή προτιμότερα προς το κέντρο της επιφάνειας. Εδώ ανοίχθηκαν 6 οπές διαμέτρου 20mm.

Αφού λοιπόν έχουμε κατασκευάσει το πρώτο κομμάτι της κατασκευής έχουμε πλέον ένα όχημα το οποίο μπορεί να σηκώσει το βάρος του, παρουσιάζει πολύ μικρές

τριβές στην κίνηση σε οποιαδήποτε κατεύθυνση, άλλα δεν έχει την δυνατότητα να κινηθεί από μονό του. Πρέπει λοιπόν να προσθέσουμε και το σύστημα της προώθησης.



Σχ.9

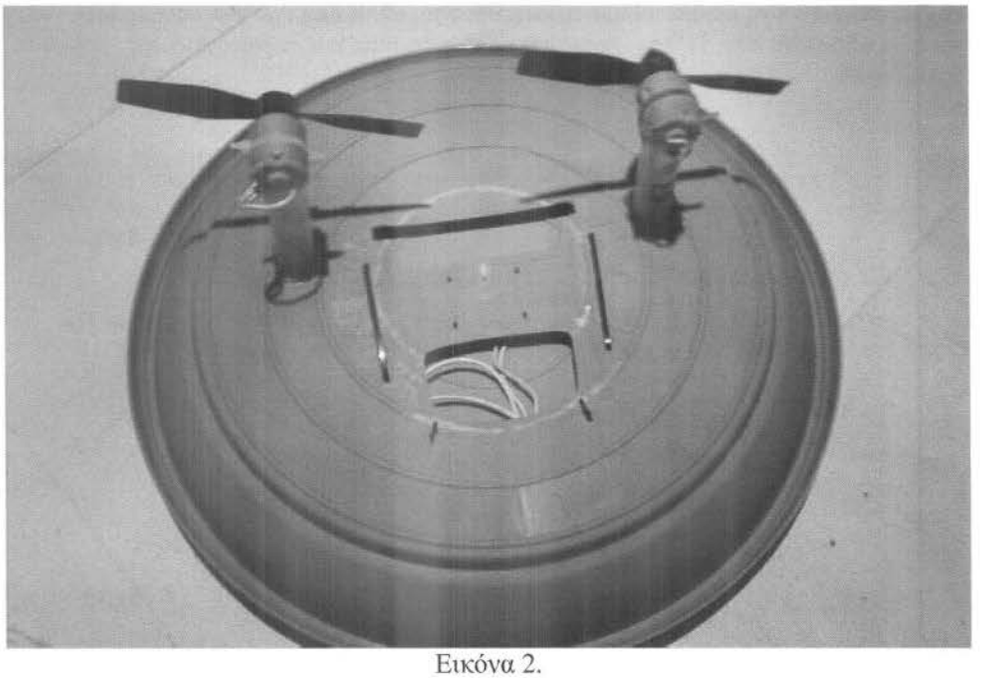


Σχ.10

Το σύστημα προώθησης

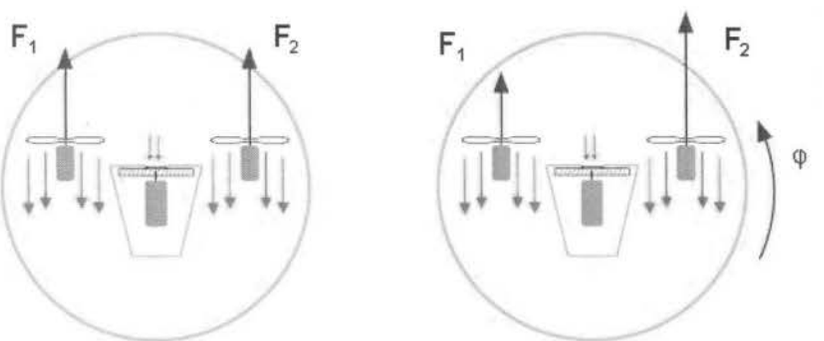
Βασικό κριτήριο για την επιλογή του υλικού για το σύστημα πρόωσης είναι και το πως θα συνδυασθεί με το σύστημα ελέγχου πορείας. Δεν αρκεί δηλαδή να τοποθετήσουμε μια έλικα που να έχει την δυνατότητα να κινεί το όχημα, άλλα πρέπει να προβλέψουμε και για το μηχανικό σύστημα που θα κάνει δυνατό τον έλεγχο θα δώσει δηλαδή ικανότητα για περιστροφή δεξιά-αριστερά.

Ως πρώτη εναλλακτική χρησιμοποιήθηκαν δυο μικρά ηλεκτρικά μοτέρ από μικρό μοντέλο αεροπλάνου με τις αντίστοιχες έλικες. Τα μοτέρ τοποθετήθηκαν παράλληλα μεταξύ τους και σε μια απόσταση 16εκ. (εικ2).



Εικόνα 2.

Έτσι όταν οι δυο κινητήρες είναι σε πλήρη τάση το όχημα προωθείται κάτω από την ώθηση που παράγουν και οι δυο έλικες συνδυασμένα. Αν όμως δώσουμε μικρότερη τάση σε ένα από τα δυο μοτέρ τότε η διαφορά στην δύναμη που θα ασκούν οι δυο έλικες θα μεταφραστεί σε ροπή στρέψης η οποία θα ξεκινήσει να περιστρέφει το όχημα γύρω από τον άξονα του δεξιά ή αριστερά ανάλογα με ποιον κινητήρα μειώνουμε την τάση (σχ11). Έτσι ελέγχοντας την τάση στους κινητήρες, ελέγχουμε και την περιστροφή της κατασκευής. Η τάση μπορεί να ελεγχθεί από έναν μικροεπεξεργαστή αρκετά εύκολα μέσω μιας κυματομορφής PWM αφού το σήμα ενισχυθεί πρώτα από ένα transistor.

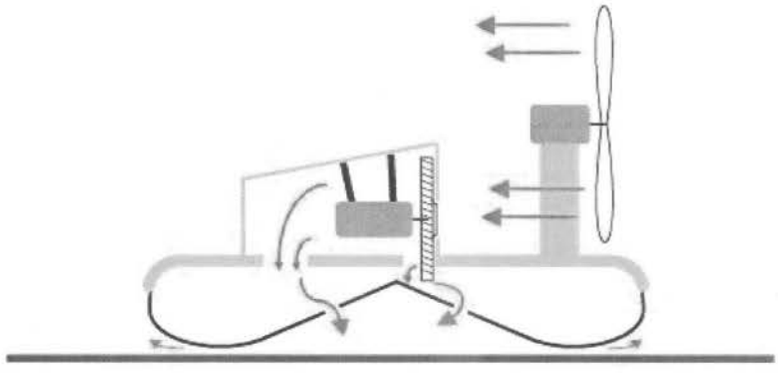


Σχ.11

Για την συγκεκριμένη εφαρμογή επιλέχθηκε το BDX53C. Η λύση αυτή έχει το μειονέκτημα ότι το σύστημα έχει πιο αργή απόκριση διότι πέρα από το καθυστέρηση του ιδίου του μηχανικού συστήματος στην περιστροφή, προστίθεται και η καθυστέρηση που έχουν οι κινητήρες από την στιγμή που θα δοθεί το σήμα έλεγχου

μέχρι να δώσουν την απαιτούμενη ώθηση που είναι η είσοδος του μηχανικού συστήματος και θα στρέψει την κατασκευή. Αυτό το πρόβλημα έγινε αντιληπτό και από πειράματα με την συγκεκριμένη λύση.

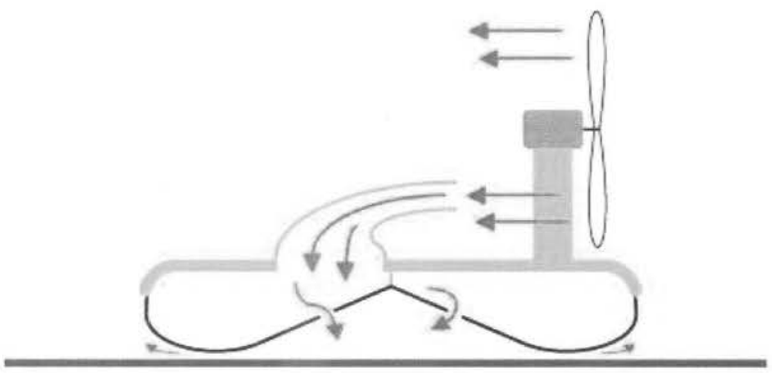
Σε αντικατάσταση λοιπών των δυο μικρών κινητήρων θα πρέπει να χρησιμοποιηθεί ένας κεντρικός, μεγαλύτερης ισχύος και να τοποθετηθούν πτερύγια εκτροπής στο πίσω μέρος που θα πραγματοποιούν τον έλεγχο. Για την επιλογή του κινητήρα έγινε έρευνα στο χώρο του μοντελισμού και επιλέχθηκε μια λύση με τα εξής χαρακτηριστικά: τάση τροφοδοσίας 6.2-8.4V, ισχύς 30W. Επιπλέον πλεονεκτήματα της επιλογής αυτής είναι ότι είναι έτοιμη λύση, με μειωτήρα στροφών, και συνδυάζεται με ένα εύρος τύπων έλικας έτσι η μέγιστη παραγόμενη ώθηση μπορεί να ρυθμισθεί από την τάση τροφοδοσίας και τον τύπο της έλικας. Για λόγους απλότητας, αλλά και σαν δείγμα της ποιότητας του ελέγχου, ο κινητήρας αυτός θα είναι μόνιμα υπό τάση και ο έλεγχος θα γίνεται μονό από τα πτερύγια. Έτσι λοιπών κατασκευάστηκε η βάση του κινητήρα και τοποθετήθηκε στο μπροστινό μέρος της κατασκευής (σχ13).



Σχ.13

Παρόλο που η επιλογή των κινητήρων καλύπτει τις ανάγκες και η κατασκευή μπορεί πλέον να εξισορροπεί το βάρος της, άλλα και να προωθείται, η χρήση του νέου, πιο δυνατού κινητήρα προώθησης έκανε πιθανή μια διαφορετική εναλλακτική που είχε απορριφθεί σε προηγούμενα στάδια. Το ερώτημα που τέθηκε ήταν κατά πόσο θα μπορούσε ο κινητήρας της κίνησης παράγει παράλληλα και την παροχή αέρα στην φούσκα για την δημιουργία της άνωσης. Τέτοιες λύσεις χρησιμοποιούνται σε πραγματικού μεγέθους hovercraft. Κατασκευάστηκε λοιπών μια ακόμα εναλλακτική κατασκευή από την οποία έλειπε το σύστημα του κινητήρα άνωσης και αντί αυτού χρησιμοποιήθηκαν δυο σωλήνες αυξανόμενης διαμέτρου, οι οποίοι εγκλωβίζουν μέρος του αέρα πίσω από την κεντρική έλικα και τον οδηγούν στην φούσκα (σχ14). Για την δοκιμή επιλέχθηκε η μεγαλύτερη τάση τροφοδοσίας 8.4V και η πιο μεγάλη από τις έλικες για το καλύτερο δυνατό αποτέλεσμα. Οι σωλήνες είναι αυξανόμενης διαμέτρου καθώς έτσι κατά μήκος τους μειώνεται η ταχύτητα κίνησης των αερίων και κατά συνέπεια αυξάνεται η πίεση. Παρόλα αυτά η άνωση που δημιουργήθηκε δεν ήταν αρκετή για να εξαλείψει της τριβές και έτσι η πιθανότητα αυτή έπρεπε να εγκαταλειφθεί. Ουσιαστικά δεν προσέφερε κάποιο συγκεκριμένο πλεονέκτημα πέρα από απλότητα στην κατασκευή, και μικρότερο βάρος. Η τροφοδοσία στην συγκεκριμένη περίπτωση έγινε με επαναφορτιζόμενες μπαταρίες τεχνολογίας λιθίου. Η επιλογή αυτή συνδυάζει πολύ μικρότερο βάρος και όγκο με μεγαλύτερη αυτονομία

και την επιθυμητή τάση των 7.2V. Έτσι αντικατέστησε την προηγούμενη μπαταρία μολύβδου.

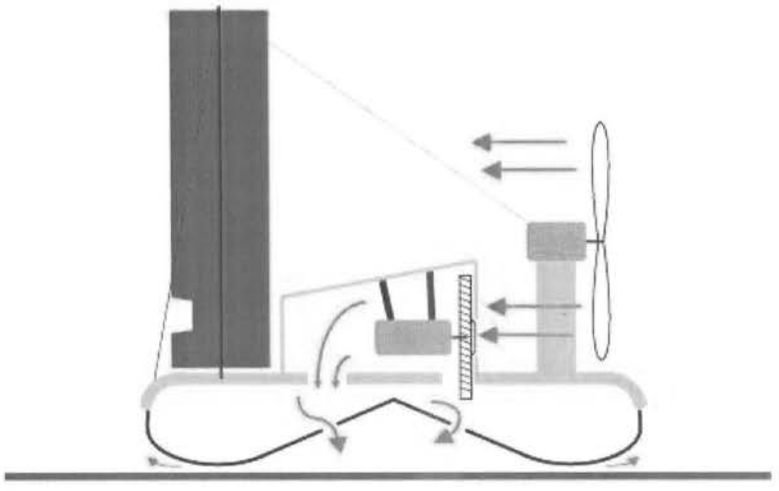


Σχ.14

Το σύστημα κατεύθυνσης

Αφού λοιπόν επιστρέψαμε στην προηγούμενη διαμόρφωση με τους δυο κινητήρες (σχ13) μένει τώρα να κατασκευαστούν και τα πτερύγια εκτροπής. Η έρευνα στην αγορά δεν έδωσε κάποια περίπτωση που θα μπορεί να καλύπτει της απαιτήσεις της κατασκευής και θα μπορεί να προσαρμοστεί εύκολα. Συνεπώς οι πτέρυγες έπρεπε να κατασκευαστούν. Σαν υλικό επιλέχθηκε τα φτερά από ένα μικρό μοντέλο αεροπλάνου που ήταν κατασκευασμένα από σκληρό αφρολέξ. Πρώτα κόπηκαν στο επιθυμητό μέγεθος δυο πτέρυγες (19.5 X 4.5) εκ. Το μέγεθος αυτό είναι απαραίτητο αν σκεφτεί κάνεις ότι η διάμετρος της έλικας είναι 18-19 εκ. Εν συνεχεία προστέθηκε στο μπροστινό τμήμα, οπού είχε γίνει η τομή, μια στρογγυλεμένη ακμή από ελαφρύ ξύλο μοντελισμού, για να γίνει πιο αεροδυναμικό το σχήμα τους. Ο στόχος είναι να τοποθετηθούν οι πτέρυγες κάθετα ώστε να μπορούν να εκτρέπουν τον αέρα δεξιά-αριστερά. Οι πτέρυγες θα πρέπει να στερεωθούν σε δυο άξονες για να είναι δυνατή η περιστροφή τους. Οι άξονες αυτοί επιλέχθηκε να περνάνε από το κέντρο των πτερύγων ώστε η πίεση του αέρα στο τμήμα μπροστά, και το τμήμα πίσω από τον άξονα να εξισορροπείται και έτσι η αντίσταση να είναι μικρότερη κατά περιστροφή τους. Αυτό απαιτεί μικρότερη δύναμη από τον ενεργοποιητή των πτερύγων, αλλά κάνει την όλη κατασκευή πιο σταθερή και αξιόπιστη. Έτσι ανοίχθηκαν δυο λεπτές τρύπες κατά μήκος των πτερύγων και στο κέντρο του πλάτους τους. Αυτή η διαδικασία ήταν αρκετά δύσκολη καθώς δεν υπήρχε στην αγορά τρυπάνι αρκετά λεπτό και ταυτόχρονα μακρύ και η όλη διεργασία έγινε στο χέρι με ικανοποιητικά τελικά αποτελέσματα ως προς την ακρίβεια. Μετά προσαρμόστηκαν, στο εσωτερικό των αξόνων που ανοίξαμε, δυο πλαστικές θήκες που θα φιλοξενήσουν τους άξονες. Οι άξονες στερεώθηκαν στην μια τους μεριά στη βάση της κατασκευής, και στην πάνω μεριά, στερεώθηκαν με σύρμα από τρία σημεία ο καθένας, ώστε να μην κινούνται προς οποιαδήποτε κατεύθυνση (σχ15). Για την στερέωση τους επέλεξα μεταλλικό σύρμα για μικρότερες ανοχές. Οι πτέρυγες είναι πλέον έτοιμες αφού ελέγχθηκε η λειτουργικότητα τους και η αντοχή τους έπρεπε να προσαρμοσθεί και το τελευταίο κομμάτι, ο μηχανισμός που θα τις ελέγχει. Για τον σκοπό αυτό έχει επιλεχθεί ένας μικρός σερβομηχανισμός από το χώρο του μοντελισμού. Το μοντέλο

είναι το DS 821, η συγκεκριμένη επιλογή θα αναλυθεί περισσότερο σε επόμενο κεφάλαιο.



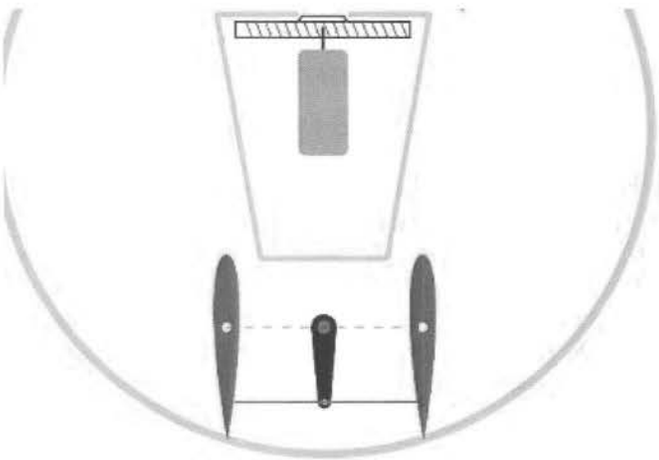
Σχ.15

Ο σερβομηχανισμός τοποθετήθηκε ανάμεσα στις δυο πτέρυγες, με τον άξονα του παράλληλο και στην ίδια ευθεία με τους άξονες των πτερύγων. Στον άξονα του σέρβο τοποθετήθηκε μια βάση πάνω στην οποία μπορεί να στηριχθεί ο άξονας ενεργοποίησης των πτερύγων. Ο άξονας ενεργοποίησης προσαρμόστηκε έτσι ώστε τα σημεία στήριξης του, στις δυο πτέρυγες και στον σερβομηχανισμό, να ισαπέχουν από τους τρεις αντίστοιχος άξονες περιστροφής (σχ16). Αυτό εξασφαλίζει ότι οι δυο πτέρυγες θα είναι παράλληλες μεταξύ τους, και η γωνία απόκλισής τους από την αρχική τους γωνία θα είναι ίση με την γωνία περιστροφής του άξονα του σέρβο. Αυτή η γωνία είναι στο σύστημα μας η ελεγχόμενη μεταβλητή και ο σερβομηχανισμός ο διεγέρτης.

Τοποθέτηση του αισθητηρίου

Ουσιαστικά το μηχανολογικό μέρος της κατασκευής μας είναι έτοιμο. Έχουμε κατασκευάσει την φούσκα και το σύστημα της άνωσης, το σύστημα προώθησης, τις πτέρυγες ελέγχου και τον ενεργοποιητή τους. Το μόνο που μένει να κάνουμε είναι να τοποθετήσουμε και το αισθητήριο. Έχουμε πει νωρίτερα ότι σαν αισθητήριο θα χρησιμοποιήσουμε ένα οπτικό ποντίκι που πρέπει να τοποθετηθεί στο κέντρο της κατασκευής και να εφάπτεται με το έδαφος. Βασικό κριτήριο για την επιλογή της συσκευής είναι η ικανότητα να λειτουργεί σε επιφάνειες πιο τραχιές απ' ό,τι συνήθως, και να έχει και μια μεγαλύτερη ανοχή στην απόσταση από την επιφάνεια λειτουργίας. Δευτερεύον κριτήριο ήταν το μέγεθος να είναι όσο το δυνατόν μικρότερο και απαραίτητη προϋπόθεση να χρησιμοποιεί το πρωτόκολλο PS2. Τα σύγχρονα ποντίκια δεν χρησιμοποιούν ποτέ μονό το πρωτόκολλο PS2 αλλά συνήθως είναι διπλού πρωτοκόλλου USB-PS2. Με βάση τα παραπάνω επιλέχθηκε μια συσκευή διπλού πρωτοκόλλου τεχνολογίας laser για μεγαλύτερο εύρος λειτουργίας. Το συνολικό μήκος της συσκευής περιορίζεται στα 97mm.

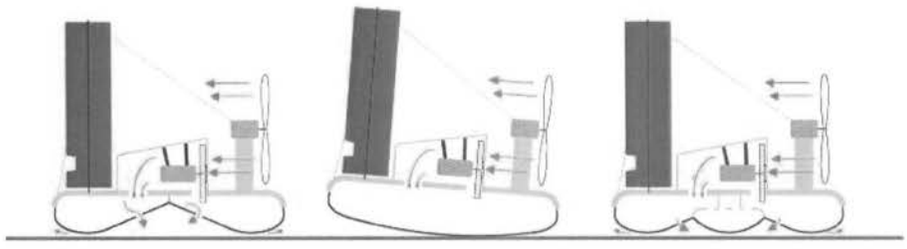
Αφού επιλέξαμε το αισθητήριο θα πρέπει να το προσαρμόσουμε στην κατασκευή μας. Το πρόβλημα στην προσαρμογή του αισθητηρίου έγκειται στο ότι δεν μπορούμε να το ενώσουμε σταθερά με την υπόλοιπη κατασκευή διότι η απόσταση από το έδαφος δεν είναι σταθερή, η φούσκα είναι εύκαμπτη και μπορεί να αλλάζει το σχήμα της κατά την λειτουργία της. Επιπλέον το ποντίκι πρέπει να εφάπτεται με το έδαφος προκειμένου να μας δίνει μετρήσεις. Έτσι εάν επιλέγαμε μια σταθερή σύνδεση η πιο μικρή ανωμαλία του εδάφους θα ήταν ικανή να σταματήσει την κίνηση ή χειρότερα να αποκολλήσει το ίδιο το αισθητήριο και το αισθητήριο δεν θα ήταν σε θέση να δίνει μετρήσεις σε όλες τις συνθήκες. Μια πρώτη σκέψη για να παρακάμψουμε αυτό το πρόβλημα είναι να φτιάξουμε μια διάταξη, που θα στηρίζει το αισθητήριο, αλλά θα του δίνει την δυνατότητα να κινείται ελεύθερα πάνω-κάτω. Να κατασκευάσουμε δηλαδή μια διάταξη ανάρτησης του αισθητηρίου. Οι πειραματισμοί που έγιναν πάνω σε αυτή την λύση δεν έδωσαν το επιθυμητό αποτέλεσμα, η κατασκευή ήταν εύθραυστη ήταν δύσκολο να ρυθμισθεί σωστά και συνολικά δεν προέκυψε κάποιο αποτέλεσμα που να λειτουργεί καλύτερα από μια σταθερή σύνδεση. Έτσι αυτή η ιδέα έπρεπε να εγκαταλειφθεί και να βρεθεί άλλη εναλλακτική για το πρόβλημα της τοποθέτησης του αισθητηρίου.



Σχ.16

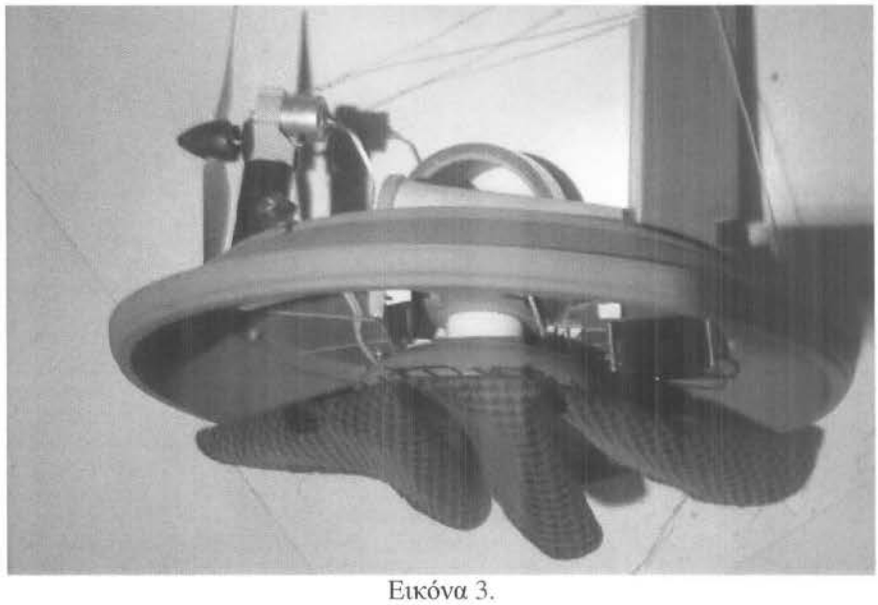
Η λύση που τελικά βρέθηκε ήταν να τοποθετήσουμε το ποντίκι πάνω στο ύφασμα της φούσκας και να μετατρέψουμε την φούσκα ώστε το κεντρικό της σημείο να εφάπτεται στο έδαφος. Κανονικά το κέντρο της φούσκας στηρίζεται ψηλά ώστε να δημιουργηθεί από κάτω το στρώμα αέρα υψηλής πίεσης που εξισορροπεί το βάρος του οχήματος. Εάν δεν γίνει αυτό το κάτω μέρος της φούσκας θα γίνει κυρτό από τη πίεση του αέρα στο εσωτερικό της και έτσι δε θα δημιουργηθεί το στρώμα αέρα που χρειαζόμαστε. Η εναλλακτική λύση που εφαρμόσαμε στην προκειμένη περίπτωση είναι να μην στηρίξουμε την φούσκα από το κέντρο της, αλλά από μια στρογγυλή περιφέρεια περίπου της μισής διαμέτρου σε σχέση με τη συνολική διάμετρο της κατασκευής. Η διάμετρος της φούσκας είναι 350mm το κάτω μέρος της στηρίζεται σε μια περιφέρεια 177mm. Αυτή η μετατροπή μας δίνει το αποτέλεσμα που φαίνεται στο σχήμα (σχ17). Βλέπουμε ότι έτσι μπορούμε να πετύχουμε και του δυο στόχους μας. Πρώτον την δημιουργία του στρώματος αέρα,και δεύτερον το κεντρικό τμήμα της φούσκας να εφάπτεται στο έδαφος ώστε να μπορούμε να τοποθετήσουμε πάνω του το αισθητήριο μας. Το γεγονός ότι το κεντρικό τμήμα εφάπτεται δεν αποτελεί

μειονέκτημα διότι με την κατάλληλη ρύθμιση δεν στηρίζεται βάρος και έτσι δεν έχει μεγάλες τριβές και εξάλλου σε κάθε περίπτωση είχαμε δεχτεί ότι το αισθητήριο μας θα εφάπτεται στο έδαφος οπότε οι λίγες παραπάνω τριβές είναι αποδεκτές. Εδώ βλέπουμε ότι η αρχική επιλογή να δώσουμε στρογγυλό σχήμα στην κατασκευή έχει ένα επιπλέον πλεονέκτημα, καθώς η συγκεκριμένη μετατροπή γίνεται πολύ πιο εύκολα σε αυτό το σχήμα παρά σε οποιοδήποτε άλλο πχ. οβάλ.



Σχ.17

Για την υλοποίηση της μετατροπής χρησιμοποιήθηκε το προστατευτικό από έναν μικρό ανεμιστήρα. Η εξωτερική του περιφέρεια ενώθηκε σταθερά με το υλικό της φούσκας και το κέντρο του στερεώθηκε με το σταθερό μέρος της κατασκευής με τέτοιο τρόπο ώστε να είναι δυνατή ρύθμιση του στο κατάλληλο ύψος (εικ3).



Εικόνα 3.

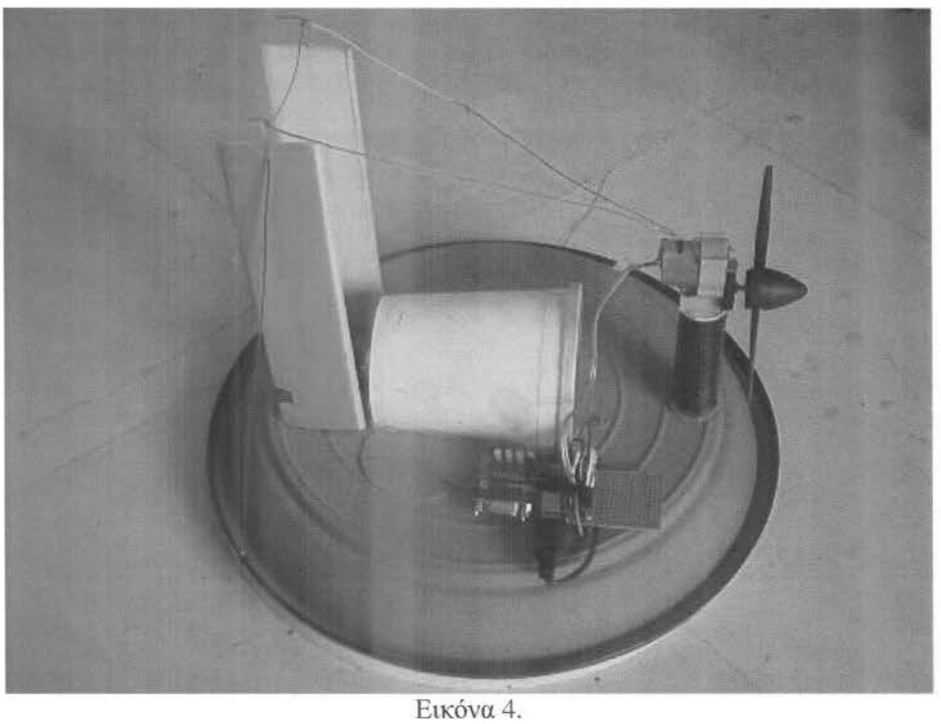
Το ποντίκι κόπηκε ώστε να μειωθούν όσο το δυνατόν οι διαστάσεις του και τοποθετήθηκε στο κέντρο της φούσκας. Οι πρώτες δοκιμές δεν δώσανε το επιθυμητό αποτέλεσμα και οι τριβές ήταν αρκετές. Αυτό οφείλεται στο γεγονός ότι η ρύθμιση στο ύψος, που έπρεπε να γίνει, ήταν τυφλή. Η φούσκα έχει διαφορετική συμπεριφορά όταν είναι σε λειτουργία στον αέρα, και όταν εφάπτεται στο έδαφος. Για να γίνει πιο ακριβής ρύθμιση η κατασκευή τέθηκε σε λειτουργία πάνω σε γυάλινη επιφάνεια ώστε να είναι ορατό το κάτω μέρος της. Με την σωστή ρύθμιση οι τριβές επανήλθαν στα

αρχικά χαμηλά επίπεδα. Η σωστή λειτουργία του ποντικιού επιβεβαιώθηκε εύκολα με την σύνδεση του σε έναν υπολογιστή με την κατασκευή σε λειτουργία.

Σε όλη την διάρκεια των πειραμάτων δόθηκε προσοχή ώστε το βάρος της κατασκευής να είναι ζυγισμένο, το κέντρο βάρους δηλαδή να βρίσκεται στο κέντρο συμμετρίας της κατασκευής. Αυτό επιτεύχθηκε με μικρή μετακίνηση των μπαταριών και την προσθήκη αντίβαρων όπου ήταν απαραίτητο. Ακόμα για την περαιτέρω μείωση των τριβών η επιφάνεια της φούσκας που μπορεί να έρθει σε επαφή με το έδαφος καλό είναι να λιπαίνεται.

Πλέον το μηχανικό κομμάτι είναι έτοιμο (εικ4). Έχουμε κατασκευάσει ένα hovercraft με όλα τα συστήματα που του επιτρέπουν την κίνηση, και έχουμε κάνει της απαραίτητες δοκιμές σε κάθε συστήματος ξεχωριστά άλλα και στη συνολική κατασκευή.

Πιο αναλυτικά κατασκευάσαμε το σασί του οχήματος, την φούσκα που του επιτρέπει να αιωρείται, το σύστημα του κινητήρα της άνοσης, το σύστημα προώθησης, το σύστημα της κατεύθυνσης με τον ενεργοποιητή και το αισθητήριο. Όλα αυτά για δουλέψουν βέβαια μαζί χρειάζονται τον ελεγκτή που θα κάνει, την επικοινωνία μεταξύ των διαφόρων στοιχείων, δυνατή και θα υπολογίζει και το κατάλληλο σήμα ελέγχου. Το ηλεκτρονικό κομμάτι θα αναλυθεί παρακάτω.



Εικόνα 4.

Ο ελεγκτής, μικροεπεξεργαστής

Αφού μελετήσαμε το μηχανικό κομμάτι της κατασκευής, θα δούμε τώρα το ηλεκτρονικό μέρος. Ο σκοπός μας είναι να αναπτύξουμε ένα σύστημα το οποίο να είναι σε θέση να εκτελεί τις παρακάτω λειτουργίες.

- Να επικοινωνεί με το αισθητήριο μας μέσο του πρωτοκόλλου PS2 και να λαμβάνει τις μετρήσεις.
- Να κάνει τους κατάλληλους μετασχηματισμούς ώστε από τις μετρήσεις αυτές να μπορεί να εξάγει την ελεγχόμενη έξοδο.
- Να υπολογίζει το κατάλληλο σήμα ελέγχου.
- Να στέλνει το σήμα αυτό στον διεγέρτη (servo)
- Να μπορεί να στέλνει δεδομένα ελέγχου και λειτουργίας σε μια εξωτερική συσκευή καταγραφής.

Είναι βέβαια σαφές ότι το σύστημα που θα μπορεί να εκτελεί όλες τις παραπάνω λειτουργίες θα είναι ψηφιακό. Ένας μικρός ψηφιακός υπολογιστής.

Το πρώτο που θα πρέπει να επιλέξουμε είναι το υλικό μας. Τι συσκευή θα χρησιμοποιήσουμε προκειμένου να πετύχουμε το ζητούμενο αποτέλεσμα. Αρχικά υπάρχουν δυο επιλογές. Η πρώτη επιλογή είναι να χρησιμοποιήσουμε ένα palmtop, έναν μικρό υπολογιστή τσέπης. Η επιλογή αυτή προσφέρει τεράστια επεξεργαστική ισχύ σε σχέση με τις απαιτήσεις μας, την έτοιμη δυνατότητα να συνδέεται με τον υπολογιστή, και υπάρχουν προγραμματιστικά περιβάλλοντα για να ελέγξουμε τις εξόδους της. Από την άλλη μεριά οι θύρες επικοινωνίας δεν είναι συμβατές με το υλικό που θα χρησιμοποιήσουμε, και θα χρειαστεί προσθήκη εξωτερικών κυκλωμάτων, η λύση έχει αυξημένο κόστος και είναι αρκετά περίπλοκη και ογκώδης.

Η εναλλακτική που χρησιμοποιήθηκε είναι ένας μικροεπεξεργαστής. Υπάρχουν στην αγορά ολοκληρωμένα που ενσωματώνουν έναν μικροεπεξεργαστή, αρκετές δυνατότητες σύνδεσης με εξωτερικό υλικό, και την δυνατότητα να προγραμματιστούν με σχετικά απλά μέσα. Η λύση αυτή προσφέρει απλότητα, μικρό όγκο και κόστος και μεγάλη ευελιξία. Μοναδικό μειονέκτημα είναι ότι το σύνολο σχεδόν του κόστους έχει να κάνει με την ανάπτυξη του προγράμματος και τον προγραμματισμό της συσκευής, έτσι είναι πολύ πιο οικονομική λύση για εφαρμογές στις οποίες θα φτιαχτούν περισσότερες από μια μονάδες.

Ο μικροεπεξεργαστής που επιλέχθηκε για αυτή την εργασία είναι ο ATmega8 της ATMEL. Είναι ένας χαμηλής κατανάλωσης, τεχνολογίας CMOS, 8-bit μικροεπεξεργαστής βασισμένος στην αρχιτεκτονική RIKC της AVR. Έχοντας την δυνατότητα να εκτελεί πολλές εντολές σε ένα κύκλο ρολογιού είναι σε θέση να πετύχει ταχύτητες μέχρι 1MIPS (Mega Instructions Per Sec) ανα 1MHz. Παρακάτω παραθέτω τα βασικά χαρακτηριστικά του μικροεπεξεργαστή.

• High-performance, Low-power AVR 8-bit Microcontroller
• Advanced RISC Architecture
– 130 Powerful Instructions – Most Single-clock Cycle Execution
– 32 x 8 General Purpose Working Registers
– Fully Static Operation
– Up to 16 MIPS Throughput at 16 MHz
– On-chip 2-cycle Multiplier

• Non-volatile Program and Data Memories
– 8K Bytes of In-System Self-Programmable Flash
Endurance: 10,000 Write/Erase Cycles
8-bit
– Optional Boot Code Section with Independent Lock Bits
In-System Programming by On-chip Boot Program
with 8K Bytes
True Read-While-Write Operation
– 512 Bytes EEPROM
Endurance: 100,000 Write/Erase Cycles In-System
– 1K Byte Internal SRAM
– Programming Lock for Software Security Programmable
• Peripheral Features
– Two 8-bit Timer/Counters with Separate Prescaler, one Compare Mode Flash
– One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
– Real Time Counter with Separate Oscillator
– Three PWM Channels ATmega8
– 8-channel ADC in TQFP and MLF package
Six Channels 10-bit Accuracy
ATmega8L
Two Channels 8-bit Accuracy
– 6-channel ADC in PDIP package
Four Channels 10-bit Accuracy
Two Channels 8-bit Accuracy
– Byte-oriented Two-wire Serial Interface
– Programmable Serial USART
– Master/Slave SPI Serial Interface
– Programmable Watchdog Timer with Separate On-chip Oscillator
– On-chip Analog Comparator
• Special Microcontroller Features
– Power-on Reset and Programmable Brown-out Detection
– Internal Calibrated RC Oscillator
– External and Internal Interrupt Sources
– Five Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, and Standby
• I/O and Packages
– 23 Programmable I/O Lines
– 28-lead PDIP, 32-lead TQFP, and 32-pad MLF
• Operating Voltages
– 2.7 - 5.5V (ATmega8L)
– 4.5 - 5.5V (ATmega8)
• Speed Grades
– 0 - 8 MHz (ATmega8L)
– 0 - 16 MHz (ATmega8)
• Power Consumption at 4 Mhz, 3V, 25°C
– Active: 3.6 mA
– Idle Mode: 1.0 mA
– Power-down Mode: 0.5 µA

Από της τρεις επιλογές εξωτερικής συσκευασίας επιλέχθηκε η πρώτη PDIP 28-pin, διότι είναι πιο διαδεδομένη, εύκολο να τοποθετηθεί σε πλακέτα και το μέγεθος δεν είναι απαγορευτικό, μόλις στα 35mm. Περισσότερες πληροφορίες για τον ATmaga8 υπάρχουν στο ΠΑΡΑΡΤΗΜΑ Α όπου βρίσκεται μέρος από το manual του κατασκευαστή.

Το επόμενο βήμα είναι ο προγραμματισμός του μικροεπεξεργαστή. Για να γίνει αυτό χρειάζονται στην ουσία δυο πράγματα : μια γλώσσα προγραμματισμού και την σύνδεση που θα φορτώσει το πρόγραμμα στον μικροεπεξεργαστή. Όσον αφορά στην πλατφόρμα προγραμματισμού υπάρχουν διαθέσιμες οι παρακάτω επιλογές. Αρχικά ο προγραμματισμός μπορεί να γίνει σε Assembly. Η πλήρης λίστα εντολών του μικροεπεξεργαστή μπορεί να βρεθεί στο site της Atmel. Σε αυτή την περίπτωση είναι απαραίτητος και ένας Assembler, ένα πρόγραμμα δηλαδή που θα μετατρέψει το πρόγραμμα γραμμένο σε assembly σε γλώσσα μηχανής και θα παράξει το αρχείο που μπορεί να φορτωθεί στον μικροεπεξεργαστή. Την δυνατότητα αυτή την παρέχει η Atmel με το AVR Studio 4.

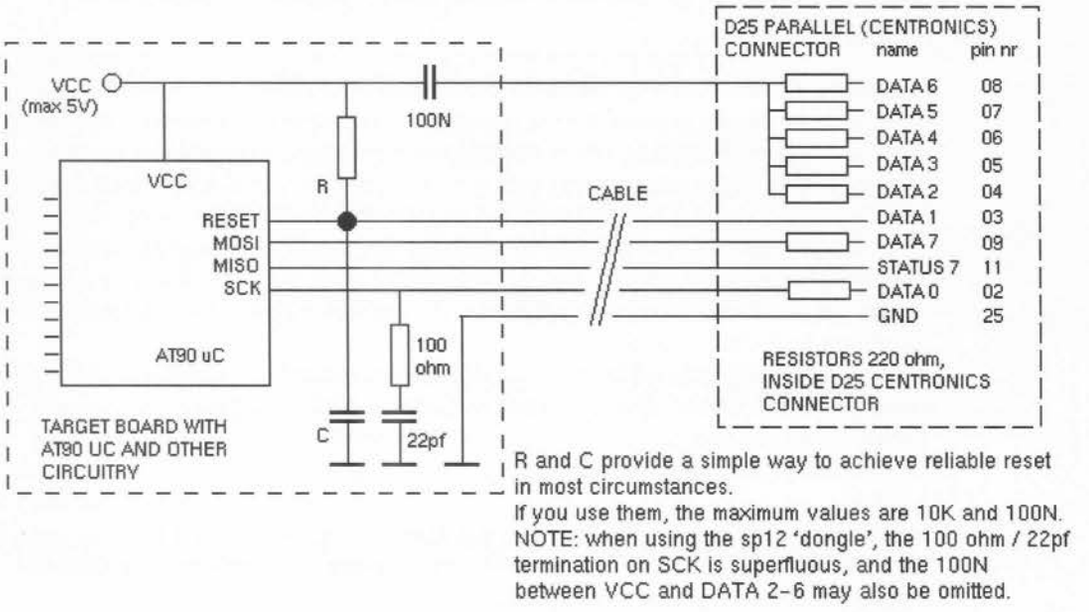
Η δεύτερη επιλογή είναι να χρησιμοποιήσουμε τη γλώσσα Basic. Αυτή την δυνατότητα την δίνουν εργαλεία όπως το Baskom AVR ή το FastAVR. Αρχικά έγιναν πειραματισμοί με το Baskom AVR. Το πρόγραμμα δίνει την δυνατότητα δημιουργίας προγράμματος μέχρι 2Kbyte στην δωρεάν εκδοχή του. Αυτό επέτρεψε να γίνουν τα πρώτα πειράματα προγραμματισμού του μικροεπεξεργαστή σε αυτό το περιβάλλον αλλά δεν ήταν αρκετό για μια λειτουργική εφαρμογή. Επιπλέον μια γλώσσα υψηλότερου επιπέδου προσφέρει περισσότερες δυνατότητες και έτσι το συγκεκριμένο πρόγραμμα εγκαταλείφθηκε.

Τέλος ο προγραμματισμός μπορεί να γίνει σε γλώσσα C. Τα εργαλεία εμπεριέχονται στο πρόγραμμα WinAVR που περιλαμβάνει και τον compiler της GNU GCC για την γλώσσα C και C++. Η τελευταία επιλογή συνδυάζει μια γλώσσα προγραμματισμού υψηλότερου επιπέδου με πολλές έτοιμες ρουτίνες, ενώ ταυτόχρονα είναι ένα πρόγραμμα ανοιχτού κώδικα που δεν προϋποθέτει οικονομικό κόστος. Μοναδικό μειονέκτημα είναι ότι χρειάζεται περισσότερη εξοικείωση στο αρχικό στάδιο μέχρι να ρυθμισθεί σωστά. Έτσι η ανάπτυξη του προγράμματος έγινε στο μεγαλύτερο μέρος σε αυτή την πλατφόρμα.

Έτσι λοιπόν δημιουργήσαμε ένα αρχείο με το πρώτο πρόγραμμα. Τα αρχεία αυτά είναι σε γλώσσα μηχανής με κατάληξη .hex. Η πρώτη δοκιμή έγινε σε γλώσσα basic και το πρόγραμμα απλά αναβόσβηνε ένα led. Αφού λοιπόν έχουμε το αρχείο πρέπει να το φορτώσομε στον μικροεπεξεργαστή. Το πρόγραμμα φορτώνεται στην μνήμη του μικροεπεξεργαστή που είναι τεχνολογίας Flash, και το πρόγραμμα δεν επηρεάζεται όταν η συσκευή είναι χωρίς τροφοδοσία. Ο μικροεπεξεργαστής έχει στη μη προγραμματιζόμενη μνήμη του βασικές ρουτίνες μεταξύ των οποίων και τις ρουτίνες που είναι απαραίτητες για τον προγραμματισμό της μνήμης Flash. Το μόνο που χρειάζεται είναι ένα πρόγραμμα που θα υλοποιεί το πρωτόκολλο επικοινωνίας και την φυσική σύνδεση με τους απαραίτητους ακροδέκτες του μικροεπεξεργαστή.

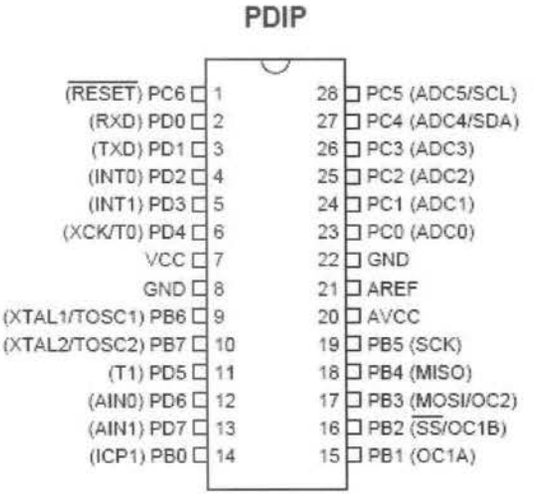
Υπάρχουν στο διαδίκτυο αρκετά προγράμματα που μπορούν να προγραμματίσουν την οικογένεια των μικροεπεξεργαστών της Atmel. Συνήθως απαιτούν ελάχιστο επιπλέον υλικό και χρησιμοποιούν την παράλληλη θύρα του υπολογιστή. Η παράλληλη θύρα δίνει τάση στους ακροδέκτες της 3.5V μεγαλύτερη από την χαμηλή τάση τροφοδοσίας των περισσότερων μικροεπεξεργαστών. Προκειμένου να παραχθεί το απαιτούμενο ρεύμα συνδέονται πολλοί ακροδέκτες μαζί στους οποίους το εκάστοτε πρόγραμμα κρατάει καθ' όλη την διαδικασία σε υψηλό δυναμικό. Εναλλακτικά βέβαια όπως στην περίπτωση μας που δεν χρησιμοποιούμε επεξεργαστή χαμηλής τροφοδοσίας πρέπει να συνδεθεί εξωτερική τροφοδοσία.

Η πρώτη προσπάθεια προγραμματισμού της συσκευής έγινε με ένα τέτοιο πρόγραμμα, συγκεκριμένα το SP12 της εταιρείας Pitronics.



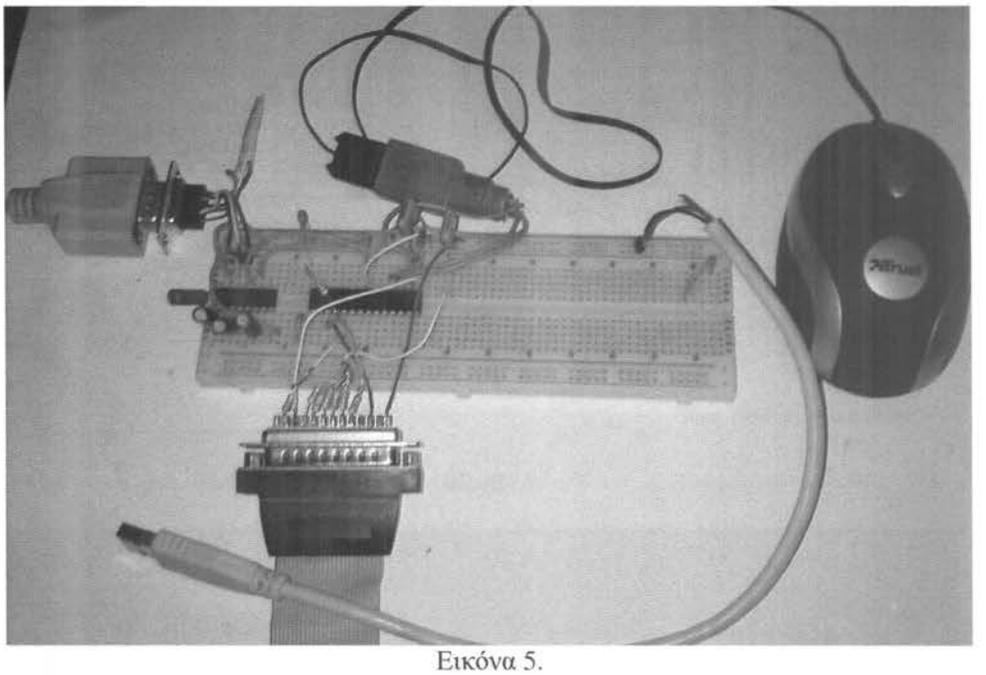
Σχ.18

Το κύκλωμα που προτείνεται για τον προγραμματισμό και υλοποιήθηκε σε breadboard φαίνεται στο σχήμα (σχ18). Σημειώστε εδώ ότι ο μικροεπεξεργαστής του σχήματος είναι διαφορετικό μοντέλο. Οι ακροδέκτες που πρέπει να γίνουν οι συνδέσεις προκύπτουν από το σχήμα (σχ19), με τους ακροδέκτες του ATmega8.



Σχ.19

Στην εικόνα 5 φαίνεται η υλοποίηση του κυκλώματος Μάζι με όλες τις περιφερειακές συνδέσεις. Το παράλληλο καλώδιο μαζί με το υλικό που απαιτείται για τον προγραμματισμό του μικροεπεξεργαστή, στα αριστερά το συριακό καλώδιο για την επικοινωνία με τον υπολογιστή, ο μετατροπέας USB to serial για την σύνδεση του ποντικιού, και το καλώδιο USB που χρησιμοποιείται εδώ σαν εξωτερική παροχή τάσης +5V. Οι περιφερειακές συνδέσεις θα αναλυθούν περισσότερο σε επόμενο κεφάλαιο.



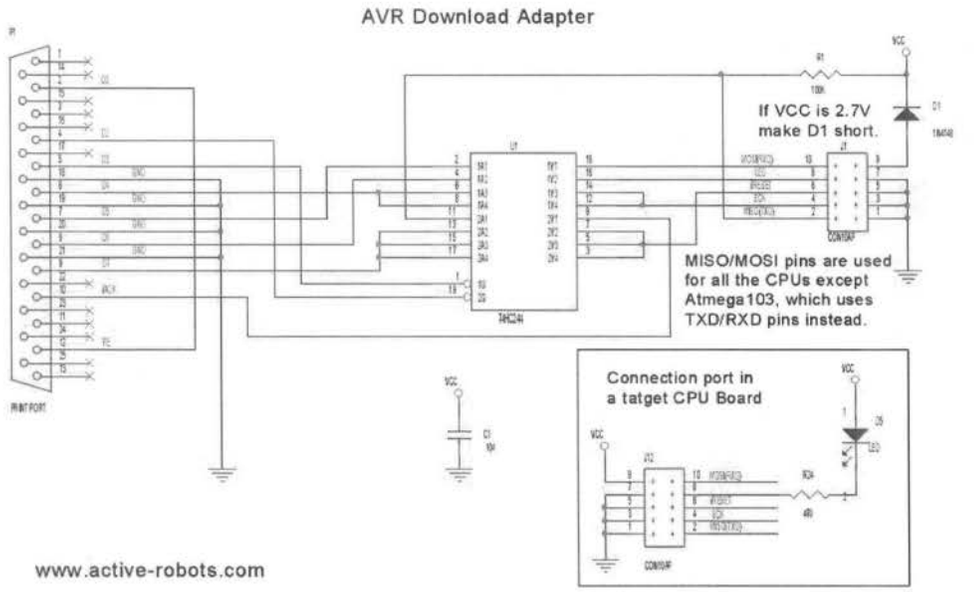
Εικόνα 5.

Δυστυχώς το συγκεκριμένο πρόγραμμα δούλεψε σωστά μονό σε περιβάλλον DOS και έτσι για τον προγραμματισμό του μικροεπεξεργαστή ήταν απαραίτητος ένας δεύτερος υπολογιστής και η μεταφορά κάθε, νέου αρχείου προγράμματος, με δισκέτα 3,5″. Η διαδικασία αυτή αν και λειτουργική, προκαλούσε καθυστερήσεις, και έτσι έγινε φανερή η ανάγκη να βρεθεί εναλλακτική λύση για τον προγραμματισμό του μικροεπεξεργαστή.

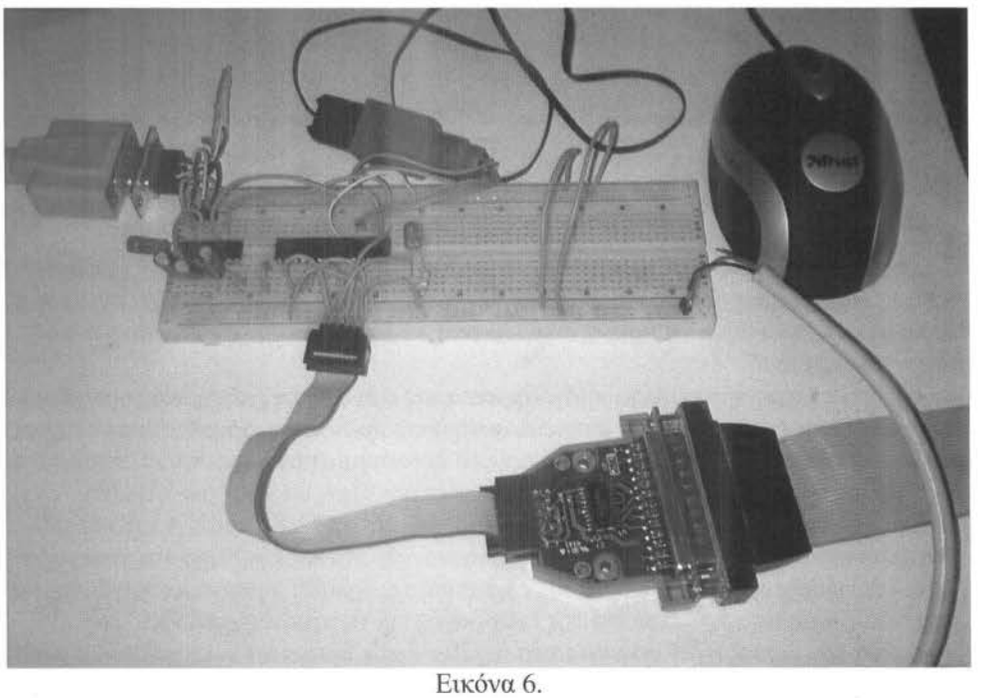
Η δεύτερη προσπάθεια προγραμματισμού έγινε με τη χρήση ενός downloader της εταιρείας active-robots. Η συσκευή αυτή είναι πολύ απλή, χρησιμοποιεί ελάχιστο υλικό και στην ουσία μιμείται έναν σειριακό προγραμματιστή της Atmel. Αυτό δίνει την δυνατότητα ο προγραμματισμός να γίνει κατευθείαν, από τον υπολογιστή ανάπτυξης του προγράμματος, μέσο του προγράμματος AVR Studio 4 της Atmel. Επιπλέον ο κονέκτορας μπορεί να ενσωματωθεί στο τελικό κύκλωμα και έτσι να δώσει την δυνατότητα για in-system programming, δηλαδή ο μικροεπεξεργαστής να μην αφαιρείται από την πλακέτα της εφαρμογής για να προγραμματιστεί. Το κύκλωμα του downloader φαίνεται στο (σχ20). Εμείς πρέπει να υλοποιήσουμε μόνο την σύνδεση που φαίνεται στο κάτω δεξιά μέρος του σχήματος.

Η υλοποίηση του κυκλώματος έγινε πρώτα σε breadboard και φαίνεται στην εικόνα 6 Ο προγραμματισμός του μικροεπεξεργαστή με αυτόν τον τρόπο είναι πρακτικός και λύνει τα προβλήματα της προηγούμενης λύσης. Παρόλα αυτά ο downloader αποδείχθηκε στην πορεία να μην είναι αρκετά αξιόπιστος και έτσι σε

συνδυασμό με την χρονική καθυστέρηση της παραγγελίας ενός καινούριου, η χρήση του απορρίφτηκε.



Σχ.20

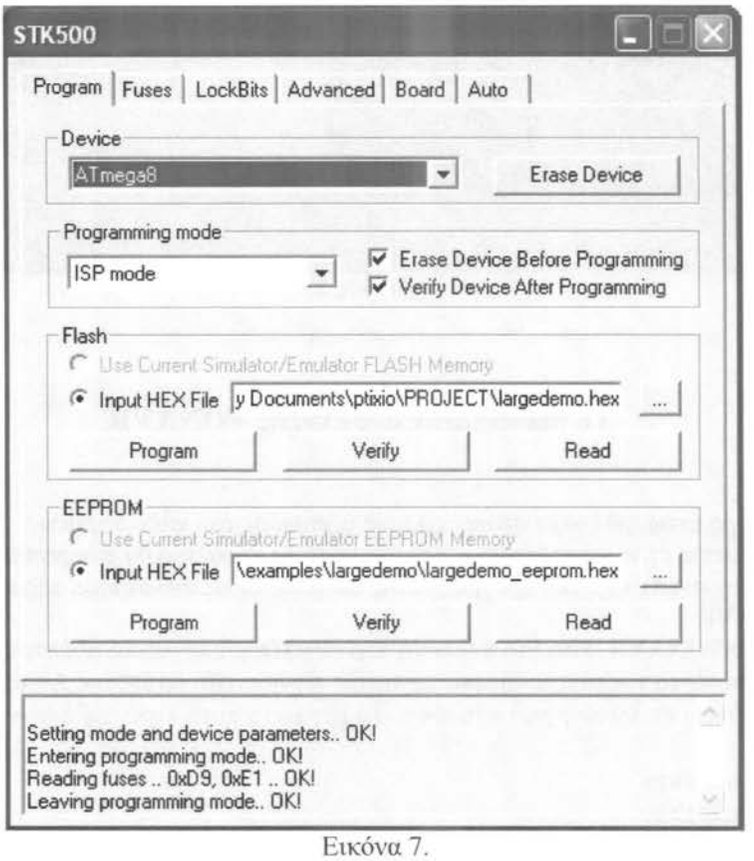


Εικόνα 6.

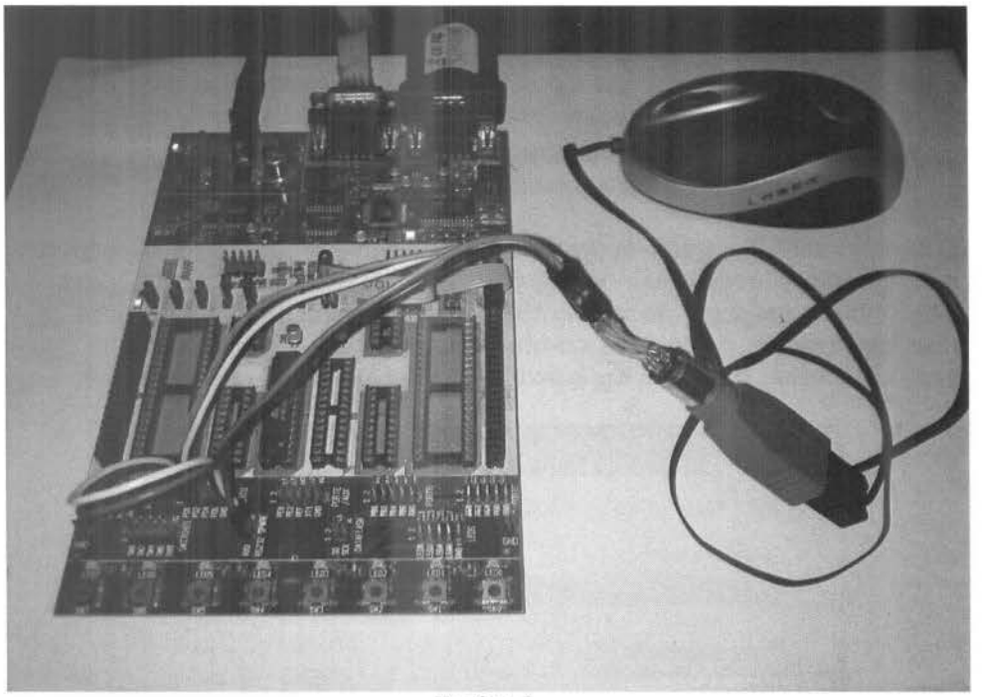
Το μεγαλύτερο μέρος της ανάπτυξης του προγράμματος και των δοκιμών έγινε τελικά με τη χρήση του kit ανάπτυξης πρωτότυπου STK 500 της Atmel. Πάρα το οικονομικό κόστος της απόκτησης, είναι ένα ολοκληρωμένο σύστημα που προσφέρει:

- Συμβατότητα με το AVR studio και βέβαια με πολλούς μικροεπεξεργαστές συμπεριλαμβανομένου του mega8.
- Σύνδεση μέσω σειριακού καλωδίου για προγραμματισμό και έλεγχο.
- Εύκολη πρόσβαση σε όλους τους ακροδέκτες του μικροεπεξεργαστή.
- 8 leds και 8 κουμπιά ενσωματωμένα για γενική χρήση.
- Επιπλέον σειριακή θήρα γενικής χρήσης.

Η σύνδεση και η διαδικασία προγραμματισμού του μικροεπεξεργαστή είναι αρκετά απλή και περιγράφεται αναλυτικά στο ΠΑΡΑΡΤΗΜΑ Β. Ενδεικτικά στις εικόνες 7 κ 8 φαίνονται οι ρυθμίσεις για την σύνδεση και τον προγραμματισμό του μικροεπεξεργαστή, καθώς και η πλακέτα με την φυσική σύνδεση για τον προγραμματισμό, την χρήση της δεύτερης σειριακής θύρας, και του ποντικιού.



Εικόνα 7.



Εικόνα 8.

Το περιβάλλον ανάπτυξης WINAVR

Μέχρι τώρα αναφερθήκαμε στους τρόπους φόρτωσης του κάθε αρχείου προγράμματος στον μικροεπεξεργαστή. Σε αυτό το κεφάλαιο θα αναφερθούμε στο περιβάλλον ανάπτυξης του προγράμματος το οποίο όπως αναφέραμε παραπάνω είναι το WINAVR.

Το WINAVR είναι μια συλλογή από εκτελέσιμα εργαλεία ανάπτυξης ανοιχτού κώδικα για τους μικροεπεξεργαστές τεχνολογίας RISK της Atmel που φιλοξενούνται σε λειτουργικό windows. Τα εργαλεία αυτά περιλαμβάνουν:

- Compilers
- Assembler
- Linker
- Librarian
- File converter
- Other file utilities
- C Library
- Programmer software
- Debugger
- In-Circuit Emulator software
- Editor / IDE
- Many support utilities

Πολλά από αυτά τα προγράμματα έχουν αναπτυχθεί σε περιβάλλον UNIX και αργότερα εισήχθησαν σε λειτουργικό windows.

Ο Compiler, είναι ο GCC ο οποίος μπορεί να χρησιμοποιηθεί για πολλούς επεξεργαστές-λειτουργικά και για διάφορες γλώσσες προγραμματισμού. Εδώ είναι διαμορφωμένος για επεξεργαστές AVR, windows, και τις γλώσσες C και C++. Έτσι αναφέρεται και ως AVR-GCC. Ο Compiler είναι απλός ένα πρόγραμμα οδηγός που μετατρέπει μια υψηλού επιπέδου γλώσσα προγραμματισμού σε Assembly, και καλεί τον assembler και τον linker αυτόματα για την δημιουργία του τελικού προγράμματος.

GNU Binutils, είναι η συλλογή των δυαδικών εργαλείων που περιλαμβάνει τον linker, ld, τον librarian ή archiver, ar, και πολλά άλλα προγράμματα που παρέχουν διαφορετικές λειτουργίες. Όλα είναι διαμορφωμένα για χρήση με τους AVR.

C Library. Η avr-libc είναι η βιβλιοθήκη για τον AVR-GCC και περιλαμβάνει πολλές από τις στάνταρ ρουτίνες της C αλλά και πολλές χρήσιμες, μη στάνταρ ρουτίνες, εξειδικευμένες για τους AVR μικροεπεξεργαστές. Περιλαμβάνει ακόμα αναλυτικό manual με πολλές πληροφορίες για την χρήση όλων των εργαλείων και παραδείγματα κώδικα. Ενδεικτικά αναφέρονται στο ΠΑΡΑΡΤΗΜΑ Γ.

Το WINAVR περιλαμβάνει υποστήριξη και εργαλεία και για την φόρτωση του προγράμματος στην μνήμη flash του μικροεπεξεργαστή, αλλά εμείς όπως είπαμε παραπάνω χρησιμοποιούμε το AVR Studio για αυτή την διαδικασία.

Make και Makefiles.

Το Make είναι το πρόγραμμα που στην ουσία κάνει τα εργαλεία (Binutils) να δουλέψουν μαζί για την δημιουργία του προγράμματος. Χρησιμοποιεί τις πληροφορίες που βρίσκονται στο Makefile για να καλέσει άλλα προγράμματα (Binutils) και να φτιάξει το πρόγραμμα. Οι πληροφορίες που βρίσκονται στο Makefile περιλαμβάνουν τον τύπο του μικροεπεξεργαστή, τη γλώσσα προγραμματισμού το τύπο του αρχείου εξόδου, optimization level, συχνότητα του επεξεργαστή, πληροφορίες για τον assembler linker debugger και πολλά άλλα. Το WINAVR περιλαμβάνει και το Mfile έναν editor για την εύκολη δημιουργία Makefiles. Για το δικό μας project χρησιμοποιήσαμε το Makefile από ένα παράδειγμα της avr-libc με τις ελάχιστες απαραίτητες αλλαγές όπως πχ τον τύπο του μικροεπεξεργαστή.

Programmers Notepad.

Το WINAVR περιλαμβάνει έναν ανοιχτού κώδικα editor για την ανάπτυξη εφαρμογών, Το PN μπορεί να καλέσει κάθε command line εργαλείο και να συλλάβει την έξοδο του. Έτσι είναι ιδανικό για να καλείται το Make το οποίο με την σειρά του καλεί όλα τα προγράμματα που είναι απαραίτητα για την δημιουργία του αρχείου προγράμματος. Το PN στη συνέχεια θα συλλάβει την έξοδο και θα την εκθέσει σε ένα παράθυρο.

Έτσι αφού έχουμε δοκιμάσει ότι το Makefile μας, λειτουργεί σωστά μπορούμε να προχωρήσομε με την σύνταξη του προγράμματος μας.

Debuging.

Ο όρος debuging είναι γενικός και περιλαμβάνει το simulation και emulation. Το WINAVR παρέχει αρκετές επιλογές για simulation και emulation. Σε αυτή την εργασία ένα μεγάλο και βασικό κομμάτι της εξέλιξης του προγράμματος αφορά στην

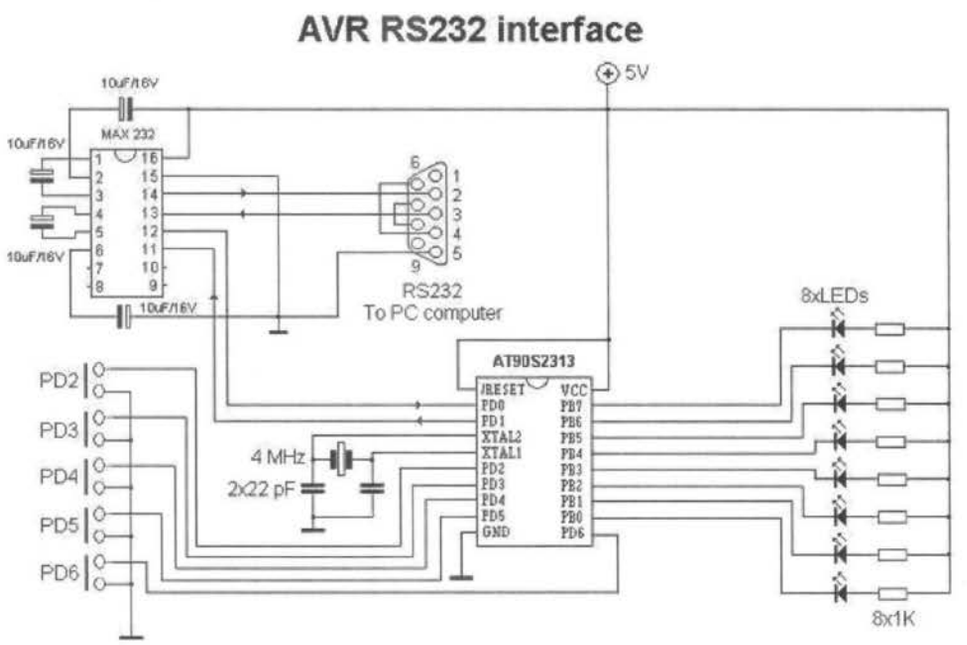
επικοινωνία του μικροεπεξεργαστή με την συσκευή PS2. Αυτό αποκλείει την χρήση simulation καθώς το επιπλέον υλικό δεν είναι δυνατό να είναι συνδεδεμένο και περιορίζει την χρηστικότητα του emulation καθώς το πρόγραμμα πρέπει να τρέχει σε πραγματικό χρόνο. Μεγάλες καθυστερήσεις π.χ. από την πλευρά του μικροεπεξεργαστή θα κάνουν τη συσκευή PS2 να απορρίπτει τα πακέτα επικοινωνίας και έτσι δεν θα μπορεί να βρεθεί κάποιο άλλο λάθος.

Εναλλακτικά για την διαδικασία του debugging χρησιμοποιήθηκε μονό η σειριακή σύνδεση του μικροεπεξεργαστή με τον υπολογιστή.

Η συριακή σύνδεση

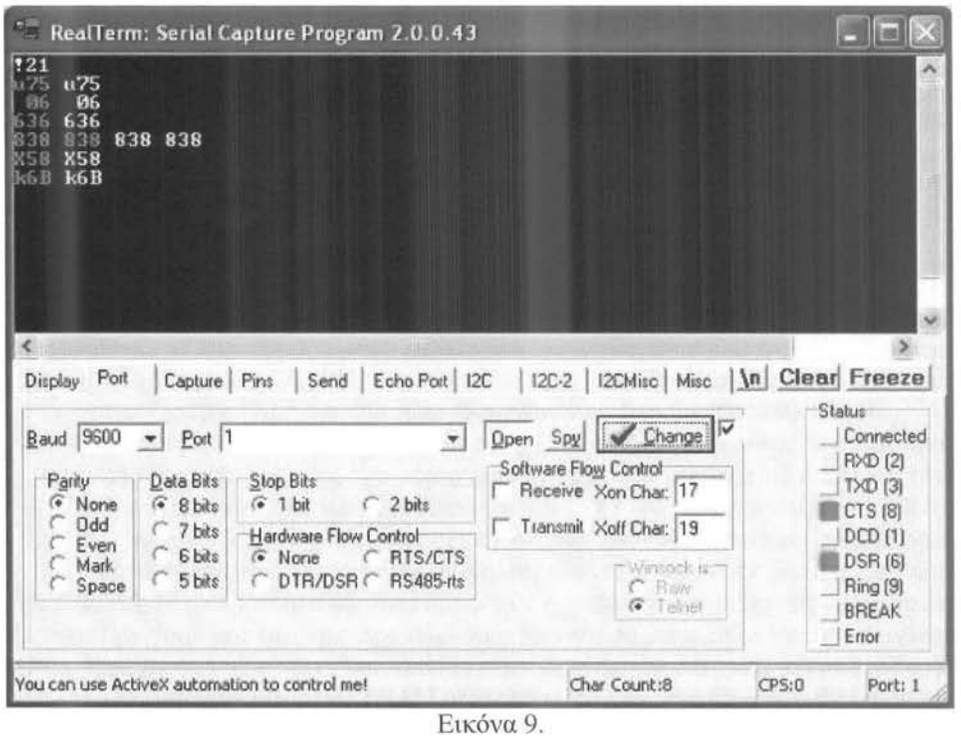
Το πρώτο βήμα για την ανάπτυξη του προγράμματος λοιπόν με βάση τα παραπάνω είναι η λειτουργία της σειριακής σύνδεσης.

Ο μικροεπεξεργαστής που χρησιμοποιούμε καθώς και οι περισσότεροι AVR έχουν ενσωματωμένο USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter). Το USART, είναι μια συσκευή συριακής επικοινωνίας, γενικής χρήσης. Και μπορεί να ρυθμιστεί ώστε να είναι συμβατή με το πρωτόκολλο RS232 συριακής επικοινωνίας του υπολογιστή. Η μοναδική προϋπόθεση για την λειτουργία είναι η προσθήκη ενός εξειδικευμένου ολοκληρωμένου μετατροπής τάσης καθώς οι ακροδέκτες του μικροεπεξεργαστή λειτουργούν στα 0-5V ενώ η σειριακή θήρα στα ±15V. Στο (σχ.21) φαίνεται ένα παράδειγμα της σύνδεσης αυτής. Στο πάνω αριστερά μέρος φαίνεται το ολοκληρωμένο MAX232 με τους τέσσερις πυκνωτές του και τις συνδέσεις προς τον AVR και το σειριακό καλώδιο. Αυτή η συνδεσμολογία υλοποιήθηκε αρχικά σε breadboard και φαίνεται στις (εικ. 5 κ 6).



Σχ.21

Στην περίπτωση βέβαια της χρήσης της πλακέτας του STK500 δεν χρειάζεται κάποιο υλικό καθώς η δεύτερη σειριακή θύρα που περιλαμβάνει, υλοποιεί ακριβώς αυτή την δυνατότητα. Από την μεριά του υπολογιστή θέλουμε να μπορούμε να βλέπουμε σε γραφικό περιβάλλον την κίνηση στη συριακή θύρα. Τη δυνατότητα αυτή μας την δίνει το πρόγραμμα Realterm. Στην (εικ. 9) φαίνεται το πρόγραμμα σε λειτουργιά. Οι επιλεγμένες ρυθμίσεις είναι Port: COM1, Baud rate: 9600 half duplex, 8 data bits, 1 stop bit.



Εικόνα 9.

Οι αντίστοιχες ρυθμίσεις πρέπει να γίνουν βέβαια και στο USART του μικροεπεξεργαστή. Το πρόγραμμα το οποίο έχει χρησιμοποιηθεί στον AVR για την πρώτη αυτή δοκιμή της συριακής πόρτας δίνεται παρακάτω:

```
//The simplest program that just receives a byte from the USART and
//transmits it back.

#include <stdint.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/io.h>


volatile uint8_t U=0;   // * note [1],[2] *

static void
putchr(char c)
{ loop_until_bit_is_set(UCSRA, UDRE);     // * note [3] *
  UDR = c;
}

// UART receive interrupt.
 ISR(USART_RXC_vect)
{
```

```
  U = UDR;
  putchr(U);
}

int
main(void)
{
  //* UART innitialaization * note [4] *
  UCSRA = _BV(U2X);      //improves baud rate error @ F_CPU = 1MHz
  UCSRB = _BV(TXEN)|_BV(RXEN)|_BV(RXCIE); // tx/rx enable, rx
//complete intr * note [5] *
  UBRRL = 103 ; /* 9600 Bd */
  TIMSK |= _BV(TOIE1);

 putchr('!');
 sei();     //enable interrupts * note [6] *

  for(;;)
 {
 }

}
```

Το παραπάνω πρόγραμμα είναι το πολύ απλό και το μόνο που κάνει είναι να παρακολουθεί την σειριακή πόρτα και όταν λάβει κάποιο πακέτο να το επιστρέφει πίσω χωρίς καμιά επεξεργασία.

Πιο αναλυτικά, αρχικά δηλώνονται οι απαραίτητες βιβλιοθήκες για τον preproccesor: δυο στάνταρ βιβλιοθήκες για τους AVR οι stdint.h και stdlib.h, η avr/interrupt.h διότι στο πρόγραμμα γίνεται χρήση της ρουτίνας εξυπηρέτησης του UART (`ISR(USART_RXC_vect)`), και η avrio.h η οποία μας επιτρέπει να χρησιμοποιούμε κατευθείαν τα ονόματα των μνημών (registers) και όχι τις διευθύνσεις τους. Ο compiler βέβαια μπορεί να κάνει την αντιστοίχηση στον εκάστοτε μικροεπεξεργαστή καθώς ο τύπος του είναι δηλωμένος στο Makefile. Στη συνέχεια του προγράμματος δηλώνεται η μοναδική μεταβλητή του προγράμματος U. Μετά οι δυο ρουτίνες του προγράμματος η putchr(char c) η οποία δέχεται ως μεταβλητή ένα χαρακτήρα, περιμένει μέχρι το UART να είναι διαθέσιμο και φορτώνει τον χαρακτήρα για να σταλεί μέσο του UART. Η δεύτερη ρουτίνα είναι η ρουτίνα εξυπηρέτησης του interrupt λήψης του UART. Όταν κάποιο πακέτο έχει σταλεί στο UART και η λήψη έχει ολοκληρωθεί με επιτυχία, η ρουτίνα αυτή εκτελείται αυτόματα διακόπτοντας την κανονική ροή του προγράμματος. Στην συγκεκριμένη περίπτωση όταν η ρουτίνα αυτή εκτελεσθεί θα πάρει απλά τον χαρακτήρα που έλαβε και θα καλέσει την ρουτίνα putchr προκειμένου να τον επιστρέψει στον υπολογιστή.

Τέλος, η κεντρική ρουτίνα void main(), η οποία με τις τέσσερις πρώτες εντολές, αρχικοποιεί το UART. Μετά στέλνει ένα θαυμαστικό ως ένδειξη ότι η αρχικοποίηση ολοκληρώθηκε με επιτυχία, ενεργοποιεί τα interrupts, και τέλος η τελευταία εντολή είναι ένα for loop χωρίς καμία μεταβλητή, ένας άπειρος κύκλος δηλαδή, από τον οποίο το πρόγραμμα δε βγαίνει ποτέ. Όλη η δουλειά του προγράμματος γίνεται πλέον από το interrupt το οποίο είναι ο μόνος τρόπος να διακοπεί ο κύκλος, στον οποιο επιστρέφει το πρόγραμμα όταν η ρουτίνα εξυπηρέτησης ολοκληρωθεί.

Σημειώσεις (notes):

1. μια μεταβλητή η οποία χρησιμοποιείται μέσα σε ISR(interrupt service routine) πρέπει να δηλώνεται ως volatile. Αυτό λέει στον compiler οτι η συγκεκριμένη μεταβλητή μπορεί να αλλάξει τιμή έκτος του πλαισίου της ανάλυσης της ροής του προγράμματος. Βλέπε ΠΑΡΑΡΤΗΜΑ Γ.

2. Ο τύπος Char είναι ένας 8-bit data type που χρησιμοποιείται από τον C compiler. Ο uint8_t είναι standard integer (8-bit unsigned) type που δηλώνεται στην βιβλιοθήκη <stdint.h>. οι δυο τύποι είναι ως εκ' τούτου συμβατοί και μια μεταβλητή που έχει δηλωθεί ως uint8_t μπορεί να χρησιμοποιηθεί από μια ρουτίνα για char.
3. Η εντολή loop_until_bit_is_clear (sfr, bit) [do { } while (bit_is_set(sfr, bit))], ορίζεται στη βιβλιοθήκη <avr/io.h> που περιλαμβάνει και την <avr/sfr_defs.h> δηλαδή για τους special function registers. Βλέπε ΠΑΡΑΡΤΗΜΑ Γ. Η λειτουργία της περιγράφεται ως εξής: Wait until bit `bit` in IO register `sfr` is clear. Η συγκεκριμένη εντολή λοιπόν ελέγχει κατά πόσο το bit UDRE (USART Data Register Empty) του register UCSRA (USART Control and Status Register A) είναι σε λογικό '1' αν δηλαδή το USART είναι έτοιμο για να δεχτεί καινούρια δεδομένα. Και περιμένει μέχρι να συμβεί αυτό. Έτσι ώστε στην επόμενη εντολή να φορτωθεί το καινούριο byte για αποστολή. Βλέπε ΠΑΡΑΡΤΗΜΑ Α.
4. Περιγραφή του USART initialization. Το USART γενικά χρησιμοποιεί έξι registers: UDR (USART Data Register) στην πράξη είναι δυο registers με την ίδια διεύθυνση, ο ένας χρησιμοποιείται για να γράφουμε τα δεδομένα προς αποστολή, και ο άλλος όταν διαβάζονται τα δεδομένα που έχουν ληφθεί. UCSRA, UCSRB, UCSRC (USART Control and Status Registers A, B and C) είναι οι τρεις registers που καταχωρούνται οι ρυθμίσεις για τον έλεγχο του USART. Και UBRRL και UBRRH (USART Baut Rate Regiaster Low and High). Μαζί αποτελούν έναν 16-bit register που χρησιμοποιείται ως prescaler για την παραγωγή της συριακής συχνότητας επικοινωνίας. Οι εντολές στο πρόγραμμα μας θέτουν λογικό '1' στα παρακάτω bits: USCSRA, U2X (Double the USART transmition speed) αυτή η ρύθμιση μειώνει τα λάθη όταν η συχνότητα του επεξεργαστή είναι στο 1Mhz. USCSRB, TXEN and RXEN (Transmitter and receiver enable) ενεργοποίηση του USART transmitter και receiver. USCSRB, RXCIE (RX complete interrupt enable) ενεργοποίηση του interrupt κατά την ολοκλήρωση λήψης, για να λειτουργήσει βέβαια το interrupt πρέπει να είναι ενεργοποιημένο και το Global Interrupt Enable στον SREG. TIMSK, TOIE1 (Timer Counter Interrupt Mask Register, Timer/Counter1 Overflow Interrupt Enable) Το USART χρησιμοποιεί για την λειτουργία του τον 16-bit Timer/Counter1 και έτσι η συγκεκριμένη ρύθμιση είναι απαραίτητη για την λειτουργία του. UBRRL = 103 Η τιμή αυτή προκύπτει από τον πίνακα στη σελίδα 158 στο ΠΑΡΑΡΤΗΜΑ Α και αντιστοιχεί σε baut rate 9600 για συχνότητα επεξεργαστή στα 8Mhz. Όλες οι υπόλοιπες ρυθμίσεις παραμένουν στις αρχικές τους τιμές οι οποίες είναι οι κατάλληλες για την λειτουργία του USART. Περισσότερες λεπτομέρειες βρίσκονται στο ΠΑΡΑΡΤΗΜΑ Α.
5. Η συγκεκριμένη εντολή όπως αναφέρθηκε παραπάνω θέτει λογικό '1' σε συγκεκριμένα bit`s αφήνοντας όλα τα υπόλοιπα στην προηγούμενη κατάσταση. Για περισσότερες λεπτομέρειες σε σχέση με την σύνταξη της εντολής ανατρέξτε στο ΠΑΡΑΡΤΗΜΑ Δ.
6. Η εντολή sei() ενεργοποιεί τα interrupts πρακτικά θέτει λογικό '1' στο bit I (Global Interrupt Enable) στον SREG Status Register.

Αφού έχουμε προγραμματίσει την συριακή πόρτα και έχουμε πρόσβαση στον μικροεπεξεργαστή μπορούμε να προχωρήσουμε στην ανάπτυξη του κώδικα για τον έλεγχο του αισθητηρίου μέσο του πρωτοκόλλου PS2. Στο επόμενο κεφάλαιο

αναφέρομαι αναλυτικά στην λειτουργία του πρωτοκόλλου και όλων των στοιχείων που πρέπει να ληφθούν υπ'όψιν για την σωστή επικοινωνία μεταξύ των δυο συσκευών, του μικροεπεξεργαστή και του ποντικιού.

Το πρωτόκολλο ps2

Σύνδεση

Οι ps2 συσκευές χρησιμοποιούν βύσματα 5-pin DIN ή 6-pin mini-DIN. Οι δυο τύποι είναι απόλυτα συμβατοί μεταξύ τους και ένας απλός μετατροπέας αρκεί για να χρησιμοποιηθεί μια συσκευή του ενός τύπου σε μια υποδοχή του άλλου. Τα 6-pin mini-DIN βύσματα έχουν πλέον επικρατήσει και χρησιμοποιούνται στην πλειονότητα των συσκευών ps2.

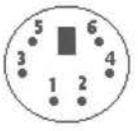
Male **Female** **5-pin DIN (AT/XT)**

(Plug)

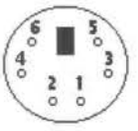
(Socket)

1 - Clock
2 - Data
3 - Not Implemented
4 - Ground
5 - Vcc (+5V)

Male **Female** **6-pin Mini-DIN (PS/2)**



(Plug)



(Socket)

1 - Data
2 - Not Implemented
3 - Ground
4 - Vcc (+5V)
5 - Clock
6 - Not Implemented

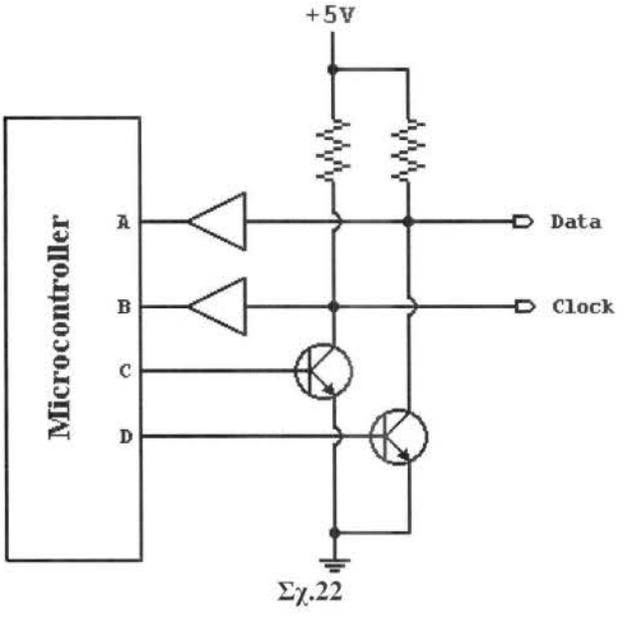
Ηλεκτρικά χαρακτηριστικά

Στο εξής θα αναφέρομαι στον υπολογιστή, ή όποια συσκευή, το πληκτρολόγιο ή το ποντίκι είναι συνδεδεμένο, ως "ελεγκτή" και στο ποντίκι ή το πληκτρολόγιο ως "συσκευή".

Τα pins Vcc/Ground τροφοδοτούν την συσκευή. Η συσκευή δεν πρέπει να καταναλώνει περισσότερα από 276mA. Αυτός είναι ο λόγος για τον οποίο δεν πρέπει να συνδέουμε μια ps2 συσκευή ενώ ο υπολογιστής είναι ενεργός, καθώς μπορεί να καεί η ασφάλεια και να αχρηστευθεί η θύρα. Αν και πλέον οι περισσότερες motherboard το υποστηρίζουν.

Οι γραμμές Data και Clock είναι ανοιχτού συλλέκτη, με pull-up αντιστάσεις στην τάση Vcc. Έχουν λοιπόν δυο καταστάσεις: κατάσταση χαμηλής (μηδενικής) τάσης, και κατάσταση υψηλής εμπέδησης. Στην χαμηλής τάσης κατάσταση ένα transistor βραχυκυκλώνει την γραμμή με τη γη (ground). Στην υψηλής εμπέδησης κατάσταση, η γραμμή λειτουργεί ως ανοιχτό κύκλωμα και συνδέεται μόνο μέσω της pull-up αντίστασης με την τάση Vcc. Έτσι όταν κανένα transistor δεν "τραβάει" την γραμμή στην γη, η τάση της εξομοιώνεται με την τροφοδοσία Vcc μέσω της pull-up αντίστασης. Το μέγεθος της αντίστασης δεν είναι πολύ σημαντικό, συνήθως 1-10 KOhms. Μεγαλύτερες αντιστάσεις δίνουν μικρότερη κατανάλωση ρεύματος ενώ οι μικρότερες πιο μικρό χρόνο ανόδου της τάσης.

Σχήμα 22: Γενική περιγραφή συνδεσμολογίας ανοιχτού συλλέκτη. Ο μικροελεγκτής διαβάζει την κατάσταση των γραμμών μέσω των ακροδεκτών A και B. Και τις ρίχνει σε χαμηλή κατάσταση δίνοντας λογικό 1 στου ακροδέκτες C και D αντίστοιχο.



Σχ.22

Στην υλοποίηση της συνδεσμολογίας ανοιχτού συλλέκτη που περιγράφεται αργότερα χρησιμοποιώ μονό έναν ακροδέκτη για κάθε γραμμή. Ενεργοποιώ την εσωτερική pull-up αντίσταση του μικροεπεξεργαστή προκειμένου να μην απαιτούνται εξωτερικές αντιστάσεις. Η γραμμή τίθεται σε κατάσταση χαμηλής τάσης θέτοντας τον ακροδέκτη ως έξοδο και την τιμή του σε λογικό μηδέν. Η γραμμή τίθεται και σε κατάσταση υψηλής εμπόδισης ορίζοντας τον ακροδέκτη σαν είσοδο.

Η επικοινωνία: γενική περιγραφή

Το πρωτόκολλο ps2 υλοποιεί μια σύγχρονη σειριακή επικοινωνία δυο δρόμων. Ο δίαυλος είναι ανενεργός όταν και οι δυο γραμμές είναι σε λογικό "1" (ανοικτού συλλέκτη). Αυτή είναι η μόνη κατάσταση στην οποία η συσκευή μπορεί να ξεκινήσει να στέλνει δεδομένα. Ο ελεγκτής έχει τον τελικό έλεγχο πάνω στον δίαυλο και μπορεί

να σταματήσει την επικοινωνία ανα πάσα στιγμή τραβώντας την γραμμή συγχρονισμού (clock) χαμηλά (0).

Η συσκευή παράγει πάντα το σήμα συγχρονισμού, αν ο ελεγκτής θέλει να στείλει δεδομένα, πρέπει πρώτα να διακόψει την επικοινωνία τραβώντας την γραμμή συγχρονισμού (clock) χαμηλά (0). Ο ελεγκτής μετά δίνει λογικό μηδέν στην γραμμή δεδομένων (data) και απελευθερώνει την γραμμή clock. Σε αυτή την κατάσταση ο ελεγκτής ζητάει να στείλει δεδομένα και η συσκευή πρέπει να ξεκινήσει να στέλνει παλμούς στην γραμμή συγχρονισμού (clock).

Κατάσταση διαύλου		
data	**clock**	**κατάσταση**
low	low	Διακοπή επικοινωνίας
low	high	Αίτηση αποστολής από τον ελεγκτή
high	low	Διακοπή επικοινωνίας
high	high	ανενεργός δίαυλος

Όλα τα δεδομένα μεταδίδονται ένα byte την φορά, μέσα σε ένα frame που αποτελείται από 11 ή 12 bits. Αυτά τα bits είναι:

- Ένα bit αρχής. Το bit αυτό είναι πάντα 0.
- 8 bits δεδομένων, το λιγότερο σημαντικό bit μεταδίδεται πρώτο.
- Ένα bit ελέγχου ισοτιμίας (odd parity bit).
- Ένα bit τέλους. Αυτό το bit είναι πάντα 1.
- Ένα bit αναγνώρισης. Αυτό το bit χρησιμοποιείτε μόνο στην επικοινωνία από τον ελεγκτή προς την συσκευή.

Το bit έλεγχου ισοτιμίας τίθεται ως "1" όταν υπάρχει άρτιος αριθμός άσων (1) στα 8 bits δεδομένων ή ως "0" όταν ο αριθμός αυτός είναι περιττός. Έτσι το άθροισμα των άσων στα 8 bit δεδομένων μαζί με το bit ισοτιμίας είναι πάντα περιττό (περιττή ισοτιμία / odd parity). Το bit αυτό χρησιμοποιείται για έλεγχο λαθών, αν η συσκευή ή ο ελεγκτής λάβει λάθος το bit ισοτιμίας ανταποκρίνεται σαν να μην έλαβε το πακέτο αυτό, δηλαδή μη επιτρεπτή εντολή.

Τα δεδομένα που στέλνονται από την συσκευή προς τον ελεγκτή διαβάζονται στην ακμή πτώσης της τάσης του παλμού συγχρονισμού. Τα δεδομένα που στέλνονται από τον ελεγκτή προς την συσκευή διαβάζονται στην ακμή ανόδου. Η συχνότητα συγχρονισμού πρέπει να βρίσκεται μεταξύ 10-16.6 kHz. Αυτό σημαίνει 30-50 microseconds υψηλή και 30-50 microseconds χαμηλή τάση για το σήμα συγχρονισμού. Στην υλοποίηση του ελεγκτή που θα περιγράψω παρακάτω η περίοδος επιλέχθηκε στα 80 microseconds. 40 microseconds υψηλή και 40 microseconds

χαμηλή τάση. Η αλλαγή ή διάβασμα της γραμμής δεδομένων γίνεται στην μέση του παλμού, δηλαδή 20 microseconds μετά την αντίστοιχη ακμή.

Η συσκευή παράγει το σήμα συγχρονισμού αλλά ο ελεγκτής έχει τον τελικό έλεγχο πάνω στην επικοινωνία. Είναι πολύ σημαντικό οι χρονικές μεταβλητές να τηρούνται αυστηρά για την σωστή λειτουργία του διαύλου.

Η επικοινωνία: από την συσκευή στον ελεγκτή

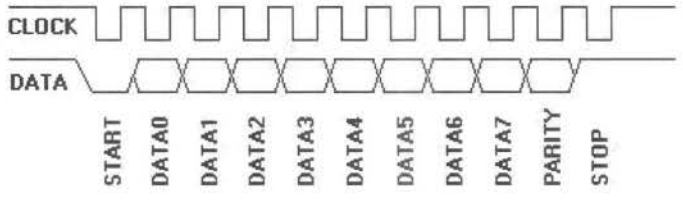
Οι γραμμές δεδομένων και συγχρονισμού (data & clock) είναι ανοιχτού συλλέκτη. Μια αντίσταση παρεμβάλλεται μεταξύ της κάθε γραμμής και της τάσης τροφοδοσίας (+5V). Έτσι η κατάσταση αδράνειας κάθε γραμμής είναι σε υψηλή τάση. Όταν η συσκευή θέλει να στείλει δεδομένα, πρώτα πρέπει να ελέγξει τη γραμμή συγχρονισμού να βρίσκεται σε λογικό "1". Εάν δεν συμβαίνει αυτό τότε ο ελεγκτής έχει διακόψει την επικοινωνία και η συσκευή πρέπει να αποθηκεύσει τα δεδομένα μέχρι ο ελεγκτής να απελευθερώσει την γραμμή συγχρονισμού (clock). Η γραμμή πρέπει να είναι συνεχόμενα σε υψηλή κατάσταση για τουλάχιστον 50 microseconds πριν η συσκευή να ξεκινήσει να στέλνει δεδομένα.

Όπως ανέφερα παραπάνω η επικοινωνία γίνεται σε frames των 11 bits τα οποία είναι:

- Ένα bit αρχής. Το bit αυτό είναι πάντα 0.
- 8 bits δεδομένων, το λιγότερο σημαντικό bit μεταδίδεται πρώτο.
- Ένα bit ελέγχου ισοτιμίας (odd parity bit).
- Ένα bit τέλους. Αυτό το bit είναι πάντα 1.

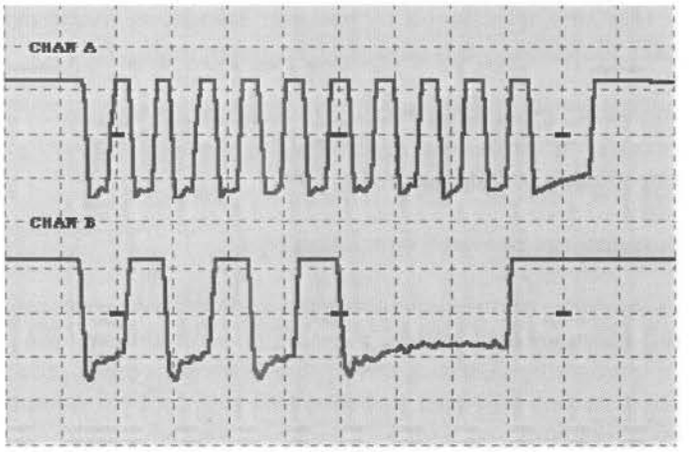
Η συσκευή "γράφει" κάθε bit στη γραμμή δεδομένων όταν η γραμμή συγχρονισμού είναι σε λογικό "1"και ο ελεγκτής το διαβάζει όταν η γραμμή συγχρονισμού είναι σε λογικό "0", όπως φαίνεται στα παρακάτω σχήματα.

Σχήμα 23: Επικοινωνία συσκευής προς ελεγκτή. Η γραμμή δεδομένων αλλάζει κατάσταση όταν η γραμμή συγχρονισμού είναι σε λογικό 1 και τα δεδομένα μπορούν να διαβαστούν όταν η γραμμή συγχρονισμού είναι σε λογικό 0.



Σχ.23

Σχήμα 24: Ο κώδικας για το γράμμα "Q" (15h) όπως στέλνεται από ένα πληκτρολόγιο. Το πρώτο κανάλι είναι η γραμμή συγχρονισμού και το δεύτερο η γραμμή δεδομένων.



Σχ.24

Η συχνότητα του ρολογιού είναι 10 έως 16 kHz ο χρόνος από την άνοδο της τάσεις του ρολογιού μέχρι την αλλαγή στη γραμμή δεδομένων πρέπει να είναι τουλάχιστον 5 microseconds και από την αλλαγή κατάστασης της γραμμής δεδομένων μέχρι την πτώση της τάσης στη γραμμή συγχρονισμού πρέπει να είναι τουλάχιστον 5 microseconds και όχι μεγαλύτερη από 25 microseconds.

Ο ελεγκτής μπορεί να διακόψει την επικοινωνία ανα πάσα στιγμή δίνοντας λογικό μηδέν στη γραμμή συγχρονισμού για τουλάχιστον 100 microseconds. Αν η επικοινωνία διακοπεί πριν τον 11° παλμό ρολογιού τότε η συσκευή πρέπει να ματαιώσει την αποστολή και να ξαναστείλει τα δεδομένα όταν απελευθερωθεί ο δίαυλος από τον ελεγκτή. Τα δεδομένα μπορεί να είναι μια εντολή ή δεδομένα μετακίνησης τα οποία μπορεί να αποτελούνται από περισσότερα του ενός byte. Έτσι η συσκευή πρέπει να ξαναστείλει όλα τα bytes από την αρχή και όχι να συνεχίσει από εκεί που διακόπηκε.

Τα δεδομένα που περιμένουν να σταλούν αποθηκεύονται στην συσκευή. Τα πληκτρολόγια κρατάνε τυπικά 16 χαρακτήρες και αγνοούν τους επόμενους χαρακτήρες μέχρι να αδειάσει η μνήμη, τα ποντίκια διατηρούνε μονό τα πιο πρόσφατα στοιχεία μετακίνησης τα οποία μπορεί να μεταβάλλονται μέχρι να αποσταλούν.

Η επικοινωνία: από τον ελεγκτή στη συσκευή

Τα δεδομένα στέλνονται με λίγο διαφορετικό τρόπο από τον ελεγκτή στη συσκευή.

Αρχικά ο παλμός συγχρονισμού παράγεται πάντα από την συσκευή έτσι ο ελεγκτής όταν θέλει να στείλει δεδομένα πρέπει να δώσει αίτηση αποστολής δεδομένων στη συσκευή ως εξής:

- Πρώτα διακόπτει την επικοινωνία δίνοντας λογικό μηδέν στη γραμμή συγχρονισμού για τουλάχιστον 100 microseconds.
- Μετά στέλνει αίτηση αποστολής δίνοντας λογικό μηδέν στην γραμμή δεδομένων και απελευθερώνοντας τη γραμμή συγχρονισμού.

Η συσκευή πρέπει να ελέγχει τον δίαυλο για αυτή την κατάσταση σε διαστήματα όχι μεγαλύτερα από 10 milliseconds. Όταν αναγνωρίσει την αίτηση αποστολής η συσκευή αρχίζει να παράγει παλμούς συγχρονισμού, 8 παλμούς δεδομένων και ένα παλμό για το bit τέλους. Ο ελεγκτής αλλάζει την κατάσταση της γραμμής δεδομένων όσο η γραμμή συγχρονισμού είναι σε χαμηλή κατάσταση και η συσκευή διαβάζει τα δεδομένα όταν είναι σε υψηλή κατάσταση. Αυτό είναι το αντίστροφο από ότι συμβαίνει στην επικοινωνία από την συσκευή στον ελεγκτή.

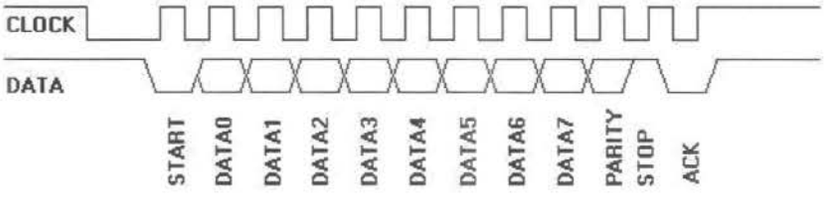
Μετά το bit τέλους η συσκευή αναγνωρίζει το byte που έλαβε ρίχνοντας την τάση στην γραμμή δεδομένων και παράγοντας ένα τελευταίο (ενδέκατο) παλμό συγχρονισμού. Αν ο ελεγκτής δεν απελευθερώσει την γραμμή δεδομένων μετά τον $11^{ο}$ παλμό συγχρονισμού, η συσκευή θα συνεχίσει να παράγει παλμούς μέχρι να απελευθερωθεί η γραμμή δεδομένων και στη συνεχεία θα δώσει κωδικό λάθους.

Ο ελεγκτής μπορεί να ματαιώσει την αποστολή δεδομένων πριν τον ενδέκατο παλμό κρατώντας την γραμμή συγχρονισμού χαμηλά για 100 microseconds.

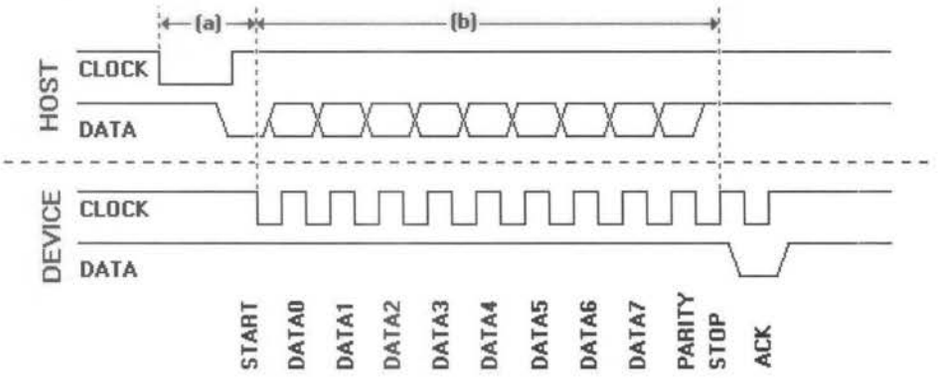
Για καλύτερη κατανόηση παραθέτω, την διαδικασία που ακολουθεί ο ελεγκτής για να στείλει δεδομένα, σε μορφή βημάτων:

1. Δίνει λογικό μηδέν στην γραμμή συγχρονισμού για 100 microseconds.
2. Δίνει λογικό μηδέν στην γραμμή δεδομένων.
3. Απελευθέρωση της γραμμής συγχρονισμού.
4. Αναμονή μέχρι η συσκευή να δώσει λογικό μηδέν στην γραμμή συγχρονισμού.
5. Γράφει στην γραμμή δεδομένων την τιμή του προτού bit.
6. Αναμονή μέχρι η συσκευή να δώσει λογικό ένα στη γραμμή συγχρονισμού.
7. Αναμονή μέχρι η συσκευή να δώσει λογικό μηδέν στη γραμμή συγχρονισμού.
8. Επανάληψη των βημάτων 5-7 για τα υπόλοιπα επτά bit δεδομένων και το bit ισοτιμίας.
9. Απελευθέρωση της γραμμής δεδομένων.
10. Αναμονή μέχρι η συσκευή να δώσει λογικό μηδέν στη γραμμή δεδομένων.
11. Αναμονή μέχρι η συσκευή να δώσει λογικό μηδέν στη γραμμή συγχρονισμού.
12. Αναμονή μέχρι η συσκευή να απελευθερώσει τις δυο γραμμές.

Το σχήμα 25 αναπαριστά γραφικά τα παραπάνω και το σχήμα 5 διαχωρίζει το διάγραμμα στα δυο, προκειμένου να φαίνεται ποια σήματα δίνονται από τον ελεγκτή και ποια από την συσκευή. Φαίνεται εδώ και η αλλαγή συγχρονισμού στο bit αναγνώρισης, όπου η αλλαγή κατάστασης, στη γραμμή δεδομένων, γίνεται όταν η γραμμή συγχρονισμού είναι σε υψηλό δυναμικό, σε αντίθεση με τα προηγούμενα 11 bits όπου η αλλαγή γίνεται όταν η γραμμή συγχρονισμού είναι σε χαμηλό δυναμικό.



Σχ.25



Σχ.26

Στο σχήμα 26 φαίνονται και δυο χρονικοί περιορισμοί που ο ελεγκτής πρέπει να ελέγχει. Το (a) δείχνει τον χρόνο που μεσολαβεί από την αρχική πτώση της τάσης στη γραμμή συγχρονισμού από την μεριά του ελεγκτή, μέχρι η συσκευή να αρχίσει την παραγωγή παλμών συγχρονισμού. Αυτός ο χρόνος δεν πρέπει να υπερβαίνει τα 15 milliseconds. Και το (b) που είναι ο χρόνος που χρειάζεται για να σταλούν τα δεδομένα, και δεν πρέπει να υπερβαίνει τα 2 milliseconds. Αν υπάρξει υπέρβαση κάποιου από τους περιορισμούς αυτούς ο ελεγκτής πρέπει να παράξει σήμα λάθους. Αμέσως μετά την λήψη του bit αναγνώρισης (επιβεβαίωσης) ο ελεγκτής μπορεί να διακόψει την επικοινωνία για να επεξεργαστεί τα δεδομένα. Αν τα δεδομένα που έστειλε απαιτούν απάντηση από την συσκευή, η απάντηση πρέπει να ληφθεί όχι αργότερα από 20 milliseconds αφότου ο ελεγκτής απελευθερώσει την γραμμή συγχρονισμού. Αν αυτό δεν συμβεί ο ελεγκτής παράγει σήμα λάθους.

Το πρωτόκολλο επικοινωνίας συσκευής κατάδειξης ps2

Υπάρχουν πολλοί τύποι συσκευών κατάδειξης: ποντίκια, trackballs, touchpads κλπ. Όλα χρησιμοποιούν ένα από τα δυο πρωτόκολλα ps2 ή USB. Παλαιότερα πρωτόκολλα συμπεριλαμβανομένου και της συριακής RS-232 έχουν πλέον εγκαταλειφθεί. Το ps2 πρωτόκολλο πρωτοεμφανίσθηκε στο "personal system/2" της IBM στα τέλη της δεκαετίας του 80. Και εξακολουθεί να είναι αρκετά διαδεδομένο παρόλο που σιγά σιγά αντικαθίσταται απο το USB.

Το πρωτόκολλο χρησιμοποιεί την δυο δρόμων σύγχρονη σειριακή επικοινωνία που περιγράφηκε παραπάνω για να στείλει δεδομένα μετακίνησης, και της κατάστασης των κουμπιών στον ελεγκτή περιφερειακών συσκευών του υπολογιστή. Ο ελεγκτής με την σειρά του στέλνει διάφορες εντολές στο ποντίκι είτε για να ορίσει την συχνότητα αποστολής δεδομένων, την ανάλυση της μετακίνησης, είτε για να επανεκκινήσει την συσκευή κλπ.

Είσοδοι, ανάλυση και κλιμάκωση μετακίνησης

Το πρωτόκολλο υποστηρίζει τις παρακάτω εισόδους: Χ μετακίνηση στα αριστερά/δεξιά, Υ μετακίνηση πάνω/κάτω, αριστερό, μεσαίο και δεξί κουμπί.
Το ποντίκι διαβάζει τις εισόδους και ανανεώνει τα δεδομένα σε αντίστοιχους καταχωρητές. Υπάρχουν αρκετές συσκευές ps2 που έχουν περισσότερες εισόδους και

χρησιμοποιούν διαφορετικούς τρόπους για να μεταδώσουν τα δεδομένα. Ένα παράδειγμα είναι το Intellimouse της Microsoft στο οποίο θα αναφερθώ επιγραμματικά παρακάτω. Οι συσκευές αυτές έχουν τις κανονικές εισόδους και επιπλέον ροδέλα μετακίνησης και δυο επιπλέον κουμπιά. Ο βασικός τύπος ποντικιού χρησιμοποιεί δυο μετρητές που καταγράφουν την μετακίνηση. Ένας για το X και ένας για το Y. Αυτοί αποτελούνται από 9 bit, τα δεδομένα που αποθηκεύονται είναι συμπληρωματικά του 2, και ο κάθε μετρητής έχει ένα ακόμα bit σαν δείκτη υπερκάλυψης (overflow flag). Τα δεδομένα των μετρητών, μαζί με την κατάσταση των τριών κουμπιών στέλνονται στον ελεγκτή σε πακέτα των τριών bytes. Οι μετρητές αντιπροσωπεύουν την μετακίνηση του ποντικιού σε σχέση με την θέση του την στιγμή που καταγράφηκε το προηγούμενο πακέτο μετακίνησης.

Όταν το ποντίκι διαβάζει τις εισόδους του καταγράφει την κατάσταση των κουμπιών και αυξάνει/μειώνει τις τιμές των μετρητών ανάλογα με την μετακίνηση σε σχέση με την προηγούμενη φορά που ανανεώθηκαν οι είσοδοι. Σε περίπτωση υπερκάλυψης ενός μετρητή, γράφεται 'ένα' στο αντίστοιχο overflow bit.

Η παράμετρος που ορίζει πόσο μεταβάλλονται οι μετρητές σε σχέση με την μετακίνηση είναι η ανάλυση. Ο ελεγκτής αλλάζει την ανάλυση χρησιμοποιώντας την εντολή "Set Resolution" (0xE8). Αν δεν οριστεί κάποια ανάλυση χρησιμοποιείται σαν αρχική ανάλυση τα 4counts/mm.

Υπάρχει μια παράμετρος που ενώ δεν επηρεάζει την τιμή των μετρητών, επηρεάζει την τιμή που στέλνεται στον ελεγκτή στο πακέτο δεδομένων. Η παράμετρος αυτή είναι η κλιμάκωση (scaling). Κανονικά η συσκευή χρησιμοποιεί scaling 1:1. Ο ελεγκτής μπορεί να επιλέξει scaling 2:1 χρησιμοποιώντας την εντολή "Set Scaling" (0xE7). Σε αυτή την περίπτωση το ποντίκι θα μετατρέψει τα δεδομένα των μετρητών πριν τα στείλει στον ελεγκτή όπως δείχνει ο παρακάτω πίνακας.

Μετρητής Μετακινήσεις	**Αναφερόμενη Μετακίνηση**
0	0
1	1
2	1
3	3
4	6
5	9
N > 5	2 * N

Η λειτουργία αυτή αναφέρεται μόνο στην κατάσταση αυτόματης αναφοράς δεδομένων από την συσκευή και δεν επηρεάζει την απόκριση της συσκευής στην εντολή: "Read Data" (0xEB).

Το πακέτο δεδομένων

Το βασικό ps2 ποντίκι στέλνει τα δεδομένα στον ελεγκτή χρησιμοποιώντας το παρακάτω πακέτο τριών bytes:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X movement							
Byte 3	Y movement							

Οι μετρητές μετακίνησης αποτελούνται από 9 bits. Το ποιο σημαντικό bit εμφανίζεται σαν πρόσημο στο πρώτο byte και τα υπόλοιπα 8 σαν ακέραιος στα δυο επόμενα bytes. Οι τιμές επηρεάζονται από την ανάλυση και κυμαίνονται από -255 έως +255.

Τρόποι λειτουργίας

Η μετάδοση των δεδομένων εξαρτάται από τον τρόπο λειτουργίας. Παρακάτω επιγραμματικά οι 4 τρόποι λειτουργίας.

- Reset - Η αρχική κατάσταση στην οποία η συσκευή πραγματοποιεί διαγνωστικά τεστ και αρχικοποίηση.
- Stream - Η κανονική κατάσταση λειτουργίας στην οποία η συσκευή στέλνει δεδομένα όταν υπάρχει μετακίνηση ή αλλαγή κατάστασης πλήκτρου.
- Remote - Ο ελεγκτής ζητάει δεδομένα από την συσκευή σε τακτά διαστήματα.
- Wrap – Μια καθαρά διαγνωστική κατάσταση λειτουργίας στην οποία η συσκευή στέλνει πίσω στον ελεγκτή τα πακέτα που δέχεται.

Κατάσταση Reset

Το ποντίκι είναι στην κατάσταση αυτή μόλις βρεθεί υπό τάση (στην εκκίνηση) ή όταν δεχθεί την εντολή "reset" (0xFF). Στην κατάσταση αυτή αρχικά τρέχει το διαγνωστικό πρόγραμμα που λέγεται BAT (Basik Assurance Test) και Θέτει τις παρακάτω αρχικές παραμέτρους:

- Sample Rate = 100 samples/sec
- Resolution = 4 counts/mm
- Scaling = 1:1
- Data Reporting = disabled

Στην συνεχεία στέλνει στον ελεγκτή ένα κωδικό, είτε 0xAA (BAT successful) είτε 0xFC (error). Και εν συνεχεία στέλνει τον κωδικό αναγνώρισης της συσκευής (Device ID) 0x00. Ο κωδικός αυτός το διαφοροποιεί από ένα πληκτρολόγιο ή από άλλου τύπου ποντίκι. Κανονικά ο ελεγκτής πρέπει να περιμένει να λάβει αυτόν τον κωδικό πριν στείλει την επόμενη εντολή αν και αυτό δεν συμβαίνει πάντα, κάποια BIOS θα στείλουν την εντολή "reset" (0xFF) αμέσως μόλις λάβουν την πρώτη απόκριση 0xAA.

Κατάσταση Stream

Στην κατάσταση αυτή το ποντίκι στέλνει δεδομένα μόλις εντοπίσει κάποια κίνηση ή αλλαγή κατάστασης κάποιου κουμπιού. Η μέγιστη συχνότητα αποστολής δεδομένων (sample rate) κυμαίνεται από 10 έως 200 δείγματα το δευτερόλεπτο. Η αρχική τιμή της παραμέτρου είναι 100 δείγματα το δευτερόλεπτο και ο ελεγκτής μπορεί να την αλλάξει χρησιμοποιώντας την εντολή "Set sample Rate" (0xF3).

Παρατηρούμε ότι στην αρχική κατάσταση έχουμε: Data Reporting: disabled. Έτσι το ποντίκι δεν θα στείλει δεδομένα μέχρις ότου ενεργοποιηθεί η παράμετρος αυτή με την εντολή: "Enable Data Reporting" (0xF4).

Η κατάσταση αυτή είναι η αρχική κατάσταση λειτουργίας και εναλλακτικά ενεργοποιείται με την εντολή "Set Stream Mode" (0xEA). Σε αυτή την κατάσταση χρησιμοποιείται η συσκευή στην εργασία αυτή.

Κατάσταση Remote

Σε αυτή την κατάσταση η συσκευή διαβάζει τις εισόδους της και ανανεώνει τους μετρητές της αλλά δεν στέλνει τα δεδομένα στον ελεγκτή παρά μόνο όταν αυτός τα ζητάει με την εντολή "Read Data" (0xEB). Μόλις λάβει την εντολή το ποντίκι θα στείλει ένα πακέτο δεδομένων και θα μηδενίσει τους μετρητές του.

Η κατάσταση αυτή δεν χρησιμοποιείται συχνά και ενεργοποιείται με την εντολή "Set Remote Mode" (0xF0).

Κατάσταση Wrap

Η συσκευή σε αυτή την κατάσταση στέλνει κάθε πακέτο που λαμβάνει πίσω στον ελεγκτή, ακόμα και τις εντολές με δυο μόνο εξαιρέσεις. Την εντολή "Reset" (0xFF) και την εντολή "Reset Wrap Mode" (0xEC).

Η κατάσταση αυτή είναι καθαρά διαγνωστική και χρησιμοποιείται σπάνια.

Οι συσκευές Intellimouse.

Μια διαδεδομένη επέκταση του κανονικού ps2 ποντικιού είναι το Intellimouse της Microsoft, το οποίο υποστηρίζει πέντε συνολικά κουμπιά και τρεις άξονες μετακίνησης (πάνω-κάτω, δεξιά αριστερά, και μια ροδέλα στον δείκτη). Για να στείλει τα επιπλέον δεδομένα χρησιμοποιεί πακέτο από 4 bytes αντί των τριών. Αλλά επειδή τα πακέτα αυτά δεν αναγνωρίζονται από τους οδηγούς/ελεγκτές των απλών ps2 ελεγκτών, η συσκευή Intellimouse πρέπει να λειτουργεί ακριβώς όπως ένα κανονικό ps2 ποντίκι μέχρι να του ζητηθεί να ενεργοποιήσει τις επιπλέον λειτουργίες. Έτσι μπορεί να λειτουργεί και με ένα απλό ps2 οδηγό έχοντας την ροδέλα και τα επιπλέον κουμπιά απενεργοποιημένα.

Η συσκευή αυτή λοιπόν λειτουργεί όπως ένα κανονικό ποντίκι μετά την αρχικοποίηση και περιμένει να λάβει την παρακάτω ακολουθία εντολών από τον ελεγκτή:

Set sample rate 200
Set sample rate 100
Set sample rate 80
Ο ελεγκτής στη συνεχεία θα στείλει εντολή "Get Device ID" (0xF2) και θα περιμένει απάντηση. Ένα κανονικό ποντίκι θα δώσει Device ID: 0x00, έτσι ο ελεγκτής θα συνεχίσει να λειτουργεί με το κανονικό ps2 ποντίκι. Εάν όμως ένα Intelllimouse είναι συνδεδεμένο τότε θα αναγνωρίσει την παραπάνω ακολουθία εντολών και θα απαντήσει device ID: 0x03. πλέον ο ελεγκτής περιμένει το ποντίκι να στέλνει δεδομένα χρησιμοποιώντας το παρακάτω πακέτο 4 bytes.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X movement							
Byte 3	Y movement							
Byte 4	Z movement							

Το Z αντιπροσωπεύει την κίνηση της ροδέλας και παίρνει τιμές από -8 έως +7. Χρησιμοποιούνται δηλαδή μόνο τα 4 λιγότερο σημαντικά bits του 4^{ου} byte. Για να ενεργοποιηθούν και τα δυο επιπλέον κουμπιά ο ελεγκτής στέλνει την παρακάτω ακολουθία εντολών.

- Set sample rate 200
- Set sample rate 200
- Set sample rate 80

Και μετά εντολή "Get Device ID" (0xF2). Αν το ποντίκι υποστηρίζει τα επιπλέον κουμπιά θα απαντήσει με Device ID: 0x04 και θα χρησιμοποιεί στο εξής το ακόλουθο πακέτο δεδομένων.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X movement							
Byte 3	Y movement							
Byte 4	Always 0	Always 0	5th Btn	4th Btn	Z movement			

Στην εφαρμογή του οδηγού συσκευής που περιλαμβάνεται στην κατασκευή παρόλο που το ποντίκι έχει ροδέλα και υποστηρίζει τις λειτουργίες Intellimouse δεν γίνεται ενεργοποίηση της δυνατότητας διότι δεν χρησιμοποιείται η ροδέλα και δεν υπάρχει λόγος για αποστολή και επεξεργασία πακέτων $4^{ων}$ bytes αντί $3^{ων}$.

Εντολές του πρωτοκόλλου Ps2

Παρακάτω είναι το σύνολο των εντολών που δέχεται ένα ps2 ποντίκι. Εάν η συσκευή στέλνει δεδομένα σε κατάσταση Stream, ο ελεγκτής πρέπει να απενεργοποιήσει την αποστολή (Disable Data Reporting 0xF5) πριν στείλει άλλη εντολή.

- 0xFF (Reset) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) και ξεκινάει την διαδικασία του Reset.
- 0xFE (Resend) - Ο ελεγκτής στέλνει την εντολή όταν λαμβάνει λάθος δεδομένα από την συσκευή αν λάβει δεύτερο λάθος πακέτο σε απάντηση στην εντολή μπορεί να στείλει και άλλη "resend" εντολή, ή να δώσει λάθος (Error 0xFC) ή να κόψει την παροχή τάσης στην συσκευή για να την επανεκκινήσει.
- 0xF6 (Set Defaults) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) και φορτώνει τις αρχικές παραμέτρους: Sample Rate = 100 samples/sec, Resolution = 4 counts/mm, Scaling = 1:1, Data Reporting = disabled.
- 0xF5 (Disable Data Reporting) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) και απενεργοποιεί την αυτόματη αποστολή δεδομένων. Η εντολή επηρεάζει την αποστολή μόνο όταν η συσκευή είναι σε κατάσταση "Stream", όπου με απενεργοποιημένη την αποστολή συμπεριφέρεται πλέον σαν να ήταν σε κατάσταση "Remote".
- 0xF4 (Enable Data Reporting) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), ενεργοποιεί την αυτόματη αποστολή δεδομένων, και μηδενίζει του μετρητές κίνησης.
- 0xF3 (Set Sample Rate) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) και περιμένει να λάβει το δεύτερο byte που είναι η συχνότητα δειγματοληψίας. Μόλις το λάβει αποκρίνεται ξανά με "acknowledge" (0xFA), μηδενίζει τους μετρητές κίνησης, και χρησιμοποιεί για συχνότητα αποστολής την τιμή που έλαβε στο δεύτερο byte. Δεκτές τιμές δειγματοληψίας είναι οι παρακάτω: 10, 20, 40, 60, 80, 100, 200 δείγματα το δευτερόλεπτο.
- 0xF2 (Get Device ID) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) ακολουθούμενο από το byte αναγνώρισης της συσκευής (0x00 για κανονικό ps2 ποντίκι). Επίσης μηδενίζει τους μετρητές κίνησης.
- 0xF0 (Set Remote Mode) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), μηδενίζει τους μετρητές και μπαίνει σε κατάσταση "Remote".
- 0xEE (Set Wrap Mode) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), μηδενίζει τους μετρητές και μπαίνει σε κατάσταση "Wrap".
- 0xEC (Reset Wrap Mode) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), μηδενίζει τους μετρητές και μπαίνει στην κατάσταση που ήταν πριν ενεργοποιηθεί η κατάσταση "Wrap".
- 0xEC (Read Data) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA) και στέλνει ένα πακέτο δεδομένων μετακίνησης. Αυτός είναι ο μονός

τρόπος να διαβαστούν τα δεδομένα στη κατάσταση "Remote". Στη συνέχεια η συσκευή μηδενίζει τους μετρητές κίνησης.

- 0xEA (Set Stream Mode) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), μηδενίζει τους μετρητές και μπαίνει σε κατάσταση "stream".
- 0xE9 (Status Request) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), μηδενίζει τους μετρητές και στέλνει το παρακάτω πακέτο τριών byte.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Byte 1	Always 0	Mode	Enable	Scaling	Always 0	Left Btn	Middle Btn	Right Btn
Byte 2	Resolution							
Byte 3	Sample Rate							

- 0xE8 (Set Resolution) Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), διαβάζει ένα ακόμα byte από τον ελεγκτή, και στέλνει ξανά "acknowledge" (0xFA). Μετά μηδενίζει τους μετρητές κίνησης, και θέτει την ανάλυση σε σχέση με το byte που έλαβε από τον ελεγκτή ως εξής:

Byte Read from Host	Resolution
00	1 count/mm
01	2 count/mm
02	4 count/mm
03	8 count/mm

- 0xE7 (Set Scaling 2:1) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), και θέτει Scaling= 2:1.
- 0xE6 (Set Scaling 1:1) - Η συσκευή αποκρίνεται στέλνοντας "acknowledge" (0xFA), και θέτει Scaling= 1:1

Οι μοναδικές εντολές που μπορεί το ποντίκι να στείλει στον ελεγκτή είναι οι: "Resend" 0xFE και "Error" 0xFC.

Αρχικοποίηση

Οι ps2 συσκευές κανονικά ανιχνεύονται κατά την διαδικασία της εκκίνησης του υπολογιστή. Δηλαδή οι ps2 συσκευές δεν μπορούν να αποσυνδεθούν, ξανασυνδεθούν με τον υπολογιστή σε λειτουργία, και ο υπολογιστής πρέπει να επανεκκινήσει σε κάθε τέτοια περίπτωση, αλλιώς μπορεί να προκληθεί ζημιά σε παλαιού τύπου motherboards.

Παρόλα αυτά τα καινούργια motherboards/λειτουργικά φαίνεται να υποστηρίζουν την διαδικασία και να λειτουργούν αξιόπιστα ακόμα και αν η συσκευή συνδεθεί αφότου εκκινήσει ο υπολογιστής.

Παρακάτω φαίνεται ένα τυπικό παράδειγμα αρχικοποίησης ενός κανονικού ps2 ποντικιού κατά την διαδικασία της εκκίνησης.

```
(Power-on Reset)
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  F3  Set Sample Rate    : Attempt to Enter Microsoft
Mouse: FA  Acknowledge        : Scrolling Mouse mode
Host:  C8  decimal 200        :
Mouse: FA  Acknowledge        :
Host:  F3  Set Sample Rate    :
Mouse: FA  Acknowledge        :
Host:  64  decimal 100        :
Mouse: FA  Acknowledge        :
Host:  F3  Set Sample Rate    :
Mouse: FA  Acknowledge        :
Host:  50  decimal 80         :
Mouse: FA  Acknowledge        :
Host:  F2  Read Device Type   :
Mouse: FA  Acknowledge        :
Mouse: 00  Mouse ID           : Response 03 if microsoft scrolling
Host:  F3  Set Sample Rate    : mouse
Mouse: FA  Acknowledge
Host:  0A  decimal 10
Mouse: FA  Acknowledge
Host:  F2  Read Device Type
Mouse: FA  Acknowledge
Mouse: 00  Mouse ID
Host:  E8  Set resolution
Mouse: FA  Acknowledge
Host:  03  8 Counts/mm
Mouse: FA  Acknowledge
Host:  E6  Set Scaling 1:1
Mouse: FA  Acknowledge
Host:  F3  Set Sample Rate
Mouse: FA  Acknowledge
Host:  28  decimal 40
Mouse: FA  Acknowledge
Host:  F4  Enable
Mouse: FA  Acknowledge
(Initialization complete)
```

Αν πατήσω το αριστερό κουμπί...

```
Mouse: 09  00001001; bit0 = Left button state; bit3 = always 1
Mouse: 00  No X-movement
Mouse: 00  No Y-movement
```

Και μετά το αφήσω...

```
Mouse: 08  00001000 bit0 = Left button state; bit3 = always 1
Mouse: 00  No X-movementt
Mouse: 00  No Y-movement
```

Παρακάτω ένα παράδειγμα με ένα Microsoft Intellimouse:

```
(Power-on Reset)
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  FF  Reset command
Mouse: FA  Acknowledge
Mouse: AA  Self-test passed
Mouse: 00  Mouse ID
Host:  F3  Set Sample Rate    : Attempt to Enter Microsoft
Mouse: FA  Acknowledge        : Scrolling Mouse mode
Host:  C8  decimal 200        :
Mouse: FA  Acknowledge        :
Host:  F3  Set Sample Rate    :
Mouse: FA  Acknowledge        :
Host:  64  decimal 100        :
Mouse: FA  Acknowledge        :
Host:  F3  Set Sample Rate    :
Mouse: FA  Acknowledge        :
Host:  50  decimal 80         :
Mouse: FA  Acknowledge        :
Host:  F2  Read Device Type   :
Mouse: FA  Acknowledge        :
Mouse: 03  Mouse ID           : Response 03 if microsoft scrolling
Host:  E8  Set Resolution     : mouse
Mouse: FA  Acknowledge
Host:  03  8 counts/mm
Mouse: FA  Acknowledge
Host:  E6  Set scaling 1:1
Dev:   FA  Acknowledge
Host:  F3  Set Sample Rate
Mouse: FA  Acknowledge
Host:  28  decimal 40
Mouse: FA  Acknowledge
Host:  F4  Enable device
```

Mouse: FA Acknowledge

Ο Driver PS2

Με βάση όλα τα παραπάνω ήμαστε έτοιμοι να δημιουργήσουμε το κομμάτι του προγράμματος που θα κάνει δυνατή την επικοινωνία μεταξύ του μικροεπεξεργαστή και της συσκευής PS2. Έναν driver στην ουσία. Το πρόγραμμα μας έχει βέβαια κάποιους περιορισμούς, αναπτύχθηκε με μια συγκεκριμένη συσκευή και προορισμός του είναι να λειτουργεί αξιόπιστα με αυτή, δε αναπτύχθηκε να προσφέρει συμβατότητα με κάθε ποντίκι, παρ' όλα αυτά με μικρές μετατροπές είναι δυνατόν να χρησιμοποιηθεί και με άλλες συσκευές και για άλλες εφαρμογές.

Καθώς το κομμάτι αυτό της ανάπτυξης του κώδικα είναι ανεξάρτητο από την υπόλοιπη εφαρμογή, και στην πορεία το μέγεθος του αποδείχθηκε αρκετά μεγάλο, θεώρησα σκόπιμο να περιληφθεί σε μια ξεχωριστή βιβλιοθήκη η οποία θα δηλώνεται στο βασικό πρόγραμμα. Παρακάτω παραθέτω το βασικό πρόγραμμα, μέχρι την δήλωση της βιβλιοθήκης ps2.c.

```
#include <stdint.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#define F_CPU 8000000UL /* CPU clock in Hertz */
#include <util/delay.h>
#include <util/parity.h>

static void
putchr(char c);

volatile uint8_t ps2buff;

#include "ps2.c"
.
.
.
```

Ότι δηλώνεται πριν την δήλωση της βιβλιοθήκης ps2.c μπορεί να χρησιμοποιηθεί από αυτή. Αυτό σημαίνει ότι η ps2.c δεν μπορεί να χρησιμοποιηθεί τελείως ανεξάρτητα. Παρόλα αυτά οι εντολές που προορίζονται για τον preprocessor μπορούν να δηλωθούν ξανά στην ps2.c ώστε να μην είναι απαραίτητες. Η ρουτίνα putchr(char c) χρησιμοποιείται για debugging και στην τελική μορφή μπορεί να παραληφθεί, και η μεταβλητή ps2buff είναι η μοναδική μεταβλητή που χρησιμοποιείται για μεταφορά δεδομένων.

Ακόμα βλέπουμε εδώ ότι δηλώνονται οι ίδιες βιβλιοθήκες που είχαμε στο προηγούμενο πρόγραμμα με την προσθήκη δυο ακόμα. Της delay.h και της parity.h που είναι απαραίτητες για συγκεκριμένες εντολές στις οποίες θα αναφερθούμε παρακάτω. Η δήλωση define ορίζει την συχνότητα λειτουργίας του μικροεπεξεργαστή και είναι απαραίτητη για την βιβλιοθήκη delay.h. τέλος η δήλωση της ps2.c είναι διαφορετική "ps2.c" και όχι <ps2.c> αυτό λέει στον preprocessor ότι η συγκεκριμένη βιβλιοθήκη βρίσκεται στον ίδιο φάκελο που είναι και το πρόγραμμα που την καλεί. Στην πραγματικότητα η ps2.c είναι ένα κομμάτι κώδικα, που ο preprocessor παρεμβάλει στο σημείο της δήλωσής της, μέσα στο βασικό πρόγραμμα. Παρακάτω παρατίθεται ο κώδικας της ps2.c.

```
#define clock      PD2
#define data       PD4



void setdata()
{DDRD &= ~_BV(data);
PORTD |= _BV(data);}
void cleardata()
{DDRD |= _BV(data);
PORTD &= ~_BV(data);}
void setclock()
{DDRD &= ~_BV(clock);
PORTD |= _BV(clock);}
void clearclock()
{DDRD |= _BV(clock);
PORTD &= ~_BV(clock);}


 void receive()
 {
 ps2buff=0;
 int f=1;
 loop_until_bit_is_set(PIND, clock);

 int i;
 for(i=0 ; i<8 ; i++)
 {
 loop_until_bit_is_clear(PIND, clock);
 _delay_us(15);
 if(bit_is_set(PIND,data))
 ps2buff |= _BV(i);     /*write 1 to i bit*/

  loop_until_bit_is_set(PIND, clock);
 }
  loop_until_bit_is_clear(PIND, clock);
 _delay_us(15);
 if(bit_is_clear(PIND,data))
 f=0;

  loop_until_bit_is_set(PIND, clock);
  loop_until_bit_is_clear(PIND, clock);
  loop_until_bit_is_set(PIND, clock);
  if (f==parity_even_bit(ps2buff))
  putchr('P');
 }

 uint8_t receive_ps2()
{
/*cli();*/
loop_until_bit_is_clear(PIND, clock);
receive();
/*sei();*/
return ps2buff;
}

void send_ps2(char b)
{

clearclock();
_delay_ms(1);

uint8_t p=0;
p=parity_even_bit(b);
cleardata();
_delay_us(20);
setclock();
```

```
int i;
for(i=0 ; i<8 ; i++)
{
loop_until_bit_is_clear(PIND, clock);
if (bit_is_clear(b,i))
cleardata();
else
setdata();

loop_until_bit_is_set(PIND, clock);

}
loop_until_bit_is_clear(PIND, clock);
if(bit_is_set(p,0))
cleardata();
else
setdata();

loop_until_bit_is_set(PIND, clock);
loop_until_bit_is_clear(PIND, clock);
setdata();
loop_until_bit_is_set(PIND, clock);
loop_until_bit_is_clear(PIND, clock);
loop_until_bit_is_set(PIND, clock);
GIFR = _BV(INTF1);
}

void ps2init()
{

/*external interupt*/
  MCUCR = _BV(ISC01); /*FALING EDGE ON INT0*/
  GICR = _BV(INT0);  /*ENABLE INT0*/

setdata();
setclock();

send_ps2(0xff); /*"Reset"*/
send_ps2(0xff); /*"Reset"*/

for(;receive_ps2()!=0xAA;) //wait until a 0xAA is successfully
//received
{
}
send_ps2(0xff); /*"Reset"*/
send_ps2(0xff); /*"Reset"*/

for(;receive_ps2()!=0xAA;) //wait until a 0xAA is succesfully
//received
{
}
receive_ps2(); /*device ID 0x00 */
send_ps2(0xe8); /*set resolution*/
receive_ps2();
send_ps2(0x03); /*8 count/mm*/
receive_ps2();
send_ps2(0xf3); /*Set Sample Rate"*/
receive_ps2();
send_ps2(0x28); /*decimal 40*/
receive_ps2();
send_ps2(0xf4); /*Enable Data Reporting"*/
receive_ps2();
//send_ps2(0xf0); /*Set Remote Mode*/
//receive_ps2();
}
```

Περιγραφή προγράμματος

Αρχικά στο πρόγραμμα δηλώνονται οι ακροδέκτες που θα χρησιμοποιηθούν με τα ονόματα clock και data. Αυτό διευκολύνει πολύ στην περίπτωση που χρειαστεί για κάποιο λόγο να χρησιμοποιηθούν διαφορετικοί ακροδέκτες, καθώς δεν χρειάζεται να αλλάξει κάθε εντολή που αναφέρεται ένας από τους δυο ακροδέκτες αλλά μονό δυοεντολές της αρχικής δήλωσης τους. Στην πραγματικότητα έγινε χρήση αυτής της δυνατότητας. Κατά την διάρκεια της εργασίας όταν πέρασα από την εξέλιξη στο κιτ STK500, στην δημιουργία της πλακέτας, κρίθηκε χρήσιμο να αλλάξουν οι ακροδέκτες. Για να γίνει αυτό πέρα από την αλλαγή των δυο αυτών εντολών το μόνο που άλλαξε ήταν το όνομα του εξωτερικού interrupt καθώς αυτό εξαρτάται από το pin στο οποίο αναφέρεται.

Στην συνέχεια ακολουθούν τέσσερις ρουτίνες setclock, clearclock, setdata και cleardata. Όπως έχω αναφέρει παραπάνω οι ακροδέκτες (clock και data) ορίζονται σαν έξοδοι όταν η τιμή τους είναι μηδέν και σαν είσοδοι όταν είναι σε λογικό ένα προκειμένου να χρησιμοποιείται η εσωτερική pull-up αντίσταση. Για να γίνει η αλλαγή από την μια κατάσταση στην άλλη χρειάζονται δυο εντολές και καθώς αυτή η αλλαγή συμβαίνει πολλές φορές μέσα στο πρόγραμμα είναι πιο συμφέρον να οριστούν ως ρουτίνες που θα καλούνται κάθε φορά που πρέπει να αλλάξει κατάσταση ένας ακροδέκτης.

Στη συνέχεια ακολουθεί η ρουτίνα receive(). Η ρουτίνα ξεκινάει την λειτουργιά της αφού η συσκευή έχει τραβήξει την γραμμή συγχρονισμού χαμηλά (clock = low). Αυτό είναι το σήμα που ενεργοποιεί και το εξωτερικό interrupt. Έτσι η ρουτίνα αυτή χρησιμοποιείται σαν πρώτη εντολή στο εξωτερικό interrupt, για να διαβάσει το πρώτο byte. Το πρώτο bit που στέλνεται από τη συσκευή είναι το start bit το οποίο η ρουτίνα το αγνοεί, δεν το διαβάζει. Μετά ακολουθεί ένα for loop, μέσα στο οποίο διαβάζονται διαδοχικά τα 8 bits δεδομένων. Ο ίδιος κύκλος επαναλαμβάνεται μια φορά για να διαβαστεί το parity bit και μια τελευταία για το stop bit ώστε να ολοκληρωθεί η αποστολή του byte. Τέλος το parity bit ελέγχεται σε αντιπαράθεση με το parity bit που υπολογίζεται από τα 8 bit δεδομένων αν προκύψει λάθος στέλνεται ο χαρακτήρας 'P' μέσω του UART, σαν ένδειξη λάθους.

Το πρόγραμμα βρίσκεται σε loops μέχρι να γίνουν οι αλλαγές στην γραμμή συγχρονισμού από την μεριά της συσκευής, και έτσι όταν διαβάζεται ένα byte ο επεξεργαστής δεν μπορεί να εκτελεί άλλες λειτουργίες. Τα δεδομένα διαβάζονται από την γραμμή δεδομένων (data) 15μsec μετά την πτώση τάσης στην γραμμή συγχρονισμού (clock).

Για τις περιπτώσεις όπου θέλουμε να διαβάσουμε ένα byte χωρίς να έχει γίνει κλήση του interrupt π.χ. να διαβάσουμε το δεύτερο και το τρίτο byte μέσα στο ίδιο interrupt ή κατά την εκκίνηση ενώ τα interrupt είναι απενεργοποιημένα, γίνεται χρήση της επόμενης ρουτίνας receive_ps2(). Η ρουτίνα αυτή περιμένει να τραβηχτεί η γραμμή συγχρονισμού χαμηλά μια φορά. Αυτό αντιστοιχεί στην ενεργοποίηση του interrupt. Και στην συνεχεία εκτελεί την receive() και επιστρέφει το byte που διαβάστηκε. Οι δυο ρουτίνες κάνουν δηλαδή την ίδια δουλειά απλά είναι διαφορετικός ο τρόπος που καλούνται και η δεύτερη επιστρέφει τα δεδομένα.

Στην συνέχεια βρίσκεται η ρουτίνα send_ps2 η οποία καλείται με μεταβλητή το προς αποστολή byte. Η ρουτίνα αρχικά δίνει μια αίτηση request to send και παράλληλα εκμεταλλεύεται τον χρόνο που μεσολαβεί από την στιγμή που θα τραβήξει σε χαμηλό δυναμικό την γραμμή συγχρονισμού μέχρι να τραβήξει σε χαμηλό δυναμικό την γραμμή δεδομένων για να υπολογίσει το parity bit. Στην

συνεχεία υπάρχει ένα for loop μέσα στο οποίο διαβάζει τους παλμούς συγχρονισμού και γράφει στη γραμμή δεδομένων κάθε ένα από τα οκτώ bit. Μετά ο κύκλος επαναλαμβάνεται για μια ακόμα φορά για το parity bit. Στην συνέχεια γράφει λογικό ένα για το stop bit, και επαναλαμβάνεται η διαδικασία μια τελευταία φορά για το acknowledge bit το οποίο δίνεται από την συσκευή, έτσι το πρόγραμμα το αγνοεί.

Τέλος ακολουθεί η ρουτίνα ps2init() στην οποία γίνεται η αρχικοποιηση της συσκευής. Η ρουτίνα αυτή χρησιμοποιεί βέβαια τις προηγούμενες ρουτίνες για να στείλει και να λάβει δεδομένα. Πιο αναλυτικά οι δυο πρώτες εντολές ρυθμίζουν το εξωτερικό interrupt ώστε να ενεργοποιείται όταν πέφτει η τάση στο pin PD2 (clock). Σημειώστε εδώ ότι τα interrupts δεν έχουν ενεργοποιηθεί ακόμα και ενεργοποιούνται μόνο στο βασικό πρόγραμμα με την εντολη sei(). Μετά η ρουτίνα στέλνει μερικές εντολές reset στη συσκευή και ελέγχει αν έλαβε την σωστή απάντηση (0xAA acknowledge). Τέλος στέλνει τις κατάλληλες εντολές για να ρυθμίσει την ανάλυση, την συχνότητα αναφοράς και την κατάσταση λειτουργίας.

Έχουμε αναφέρει ότι η συσκευή μας είναι διπλού πρωτοκόλλου ps2/USB παρόλα αυτά εμείς δεν χρειάζεται να κάνουμε τίποτα για αυτό μόλις στείλουμε την πρώτη εντολή η συσκευή θα αναγνωρίσει το χρησιμοποιούμενο πρωτόκολλο και θα μπει στην αντίστοιχη κατάσταση λειτουργίας για PS2. Ακόμα η συσκευή έχει ροδέλα πράγμα που την κάνει συμβατή με την λειτουργία Intellimouse. Η συγκεκριμένη λειτουργία όμως είναι άχρηστη για την εφαρμογή μας και θα επιβάρυνε κάθε πακέτο δεδομένων με ένα τέταρτο byte. Έτσι ο ελεγκτής, δεν στέλνει ποτέ τις εντολές, που θα ενεργοποιούσαν αυτή την λειτουργία. Οι ρυθμίσεις που επιλέξαμε για το ποντίκι μας εδώ είναι resolution: 8 counts/mm, sample rate: 200Hz, Stream mode. Η κατάσταση λειτουργίας Stream επιλέχθηκε για δυο λόγους πρώτα δεν χρειάζεται να σταλεί η εντολή για να διαβαστούν τα δεδομένα και αυτό απελευθερώνει χρόνο επεξεργασίας. Και δεύτερον επειδή η συσκευή στέλνει δεδομένα σε τακτά χρονικά διαστήματα μπορεί να χρησιμοποιηθεί αυτό για τον συγχρονισμό του βασικού προγράμματος. Αυτό γίνεται πιο σημαντικό με την παράλληλη λειτουργιά της συριακής πόρτας η οποία δεσμεύει τον μοναδικό 16-bit timer του μικροεπεξεργαστή. Και αφήνει ως μόνη άλλη εναλλακτική την χρήση 8-bit timer σε συνδυασμό με software implementation για τον χρονισμό του βασικού προγράμματος.

Το κυρίως πρόγραμμα

Ας περάσουμε τώρα στην περιγραφή του κυρίως προγράμματος. Ακολουθεί ο κώδικας:

```
      #include <stdint.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#define F_CPU 8000000UL /* CPU clock in Hertz */
#include <util/delay.h>
#include <util/parity.h>
#include <math.h>

/*  functions declaration*/

static void
```

```
putchr(char c);


#include "ps2.c"

/*variable definition*/
volatile uint8_t countm=0;
volatile uint8_t countservo=4;
volatile uint8_t U=100;
volatile int x=0;
volatile int y=0;



/* UART receive interrupt. */
 ISR(USART_RXC_vect)
{
  U = UDR;
}


  /*Send character c down the UART Tx.*/
static void
putchr(char c)
{ loop_until_bit_is_set(UCSRA, UDRE);
  UDR = c;
}



ISR(TIMER2_COMP_vect)
{
PORTC &= ~_BV(PC3);/*CLEAR PC3*/
TIMSK &= ~_BV(OCIE2);
}

ISR(INT0_vect)/*clock low*/
 {

uint8_t byte1;
uint8_t byte2;
uint8_t byte3;


 byte1=receive();
 byte2=receive_ps2();
 byte3=receive_ps2();

 x+=byte2;
 y+=byte3;

 if bit_is_set(byte1,4)
{x-=256;
}
if bit_is_set(byte1,5)
{y-=256;
}
countm++;
countservo++;

if (countservo==5)
{
countservo=0;

PORTD |= _BV(PORTD6);/*SET PD6*/

_delay_ms(1);
```

```
uint8_t i=0;
for(i=0;i<U;i++)
_delay_us(5);

PORTD &= ~_BV(PORTD6);/*CLEAR PD6*/
}

GIFR |= _BV(INTF0);/*clear int0*/
}

void uart_init()
{
  /* UART */
  UCSRA = _BV(U2X);     /*improves baud rate error @ F_CPU = 1 MHz */
  UCSRB = _BV(TXEN)|_BV(RXEN)|_BV(RXCIE); // tx/rx enable,
 // Rx complete intrrupt
  UBRRL = 103 ; /* 9600 Bd */
  TIMSK |= _BV(TOIE1);
}

int
main(void)
{/* Do all the startup-time peripheral initializations.*/

 TCCR2 |= _BV(WGM21)|_BV(CS22)|_BV(CS20);//TIMER COUNTER2

 uart_init();

 DDRB |= _BV(DDB0);
 PORTB |= _BV(PB0);

 DDRD |= _BV(DDD6);//SERVO PIN
 DDRC |= _BV(DDC3);//LED PIN
 PORTC |= _BV(PC3);


  ps2init();

  putchr('!');       /*initialization complete!*/

  GIFR |= _BV(INTF0);   /*clear int0*/
  sei();                 /*enable interrupts */

float Ed=0;       //previus error store
float Ei=0;       //integral error
float Sp=0;       //set point

int mode=0;
int countmode=0;

float f;
float pid;
int Kp=135;
int Kd=15;
int Ki=0;


  for(;;)            /*loop forever*/
  {

  if(countm==20)
  {

int xi=x;
int yi=y;

if (mode==0) //straight line
{
```

```
Sp=0;
}

if (mode==1) //left turn
{
Sp=M_PI/2;
}

if (mode==2) //right turn
{
Sp=-M_PI/2;
}

if (mode==3) //straight again
{
mode=0;
Sp=0;
}

x=0;
y=0;

countm=0;
countmode++;

if (countmode==2)
PORTC &= ~_BV(PC3);/*CLEAR PC3*/

if (countmode==20)
{
countmode=0;
mode++;
PORTC |= _BV(PC3);/*SET PC3*/
}


f=atan2(xi,yi);

f=Sp-f; //error=set point-current angle

if (f>M_PI)
f-=2*M_PI;

if (f<M_PI)
f+=2*M_PI;

if (f<-2)
f=-2;

if (f<-2)
f=-2;       //limit f to +2, -2

//PID
Ed=f-Ed;   //dif. Error=error-preveus error
Ei+=f;
if (Ei>10)
Ei=10;

if (Ei<-10)
Ei=-10;           //limit Ei to +10, -10

pid=f*Kp;
pid+=Ed*Kd;
pid+=Ei*Ki;

Ed=f;

if (pid>100)
pid=100;
```

```
if (pid<-100)
pid=-100;          //limit pid to +100, -100

U=pid;
putchr(U);
U+=100;
  }
  }

}
```

Όπως έχω αναφέρει παραπάνω, το πρόγραμμα αρχίζει με την δήλωση των βιβλιοθηκών που αντιστοιχούν στις εντολές που χρειάσθηκαν για την υλοποίηση του προγράμματος. Επίσης δηλώνεται και η βιβλιοθήκη ps2.c ώστε οι ρουτίνες που μελετήσαμε παραπάνω να είναι διαθέσιμες για χρήση στο κυρίως πρόγραμμα.

Αμέσως μετά δηλώνονται οι μεταβλητές που ανανεώνονται από τα interrupts, ως volatile. Και στην συνέχεια ακολουθούν οι ρουτίνες του προγράμματος. Ας αρχίσουμε την περιγραφή από την τελευταία και βασική ρουτίνα: main. Εδώ πρώτα αρχικοποιείται ο timer counter 2. Ο timer αυτός χρησιμοποιείται για τον έλεγχο του Led και έχει μοναδικό σκοπό το debugging, δεν είναι βασικό μέρος της λειτουργίας του προγράμματος. Αμέσως μετά καλείται η ρουτίνα uartinit που αρχικοποιεί την σειριακή σύνδεση όπως είδαμε στο πρώτο πρόγραμμα serial_test. Στη συνέχεια ορίζονται και αρχικοποιούνται οι ακροδέκτες που χρησιμοποιούμαι για να ελέγξουμε το servo και το Led. Μετά καλείται η ρουτίνα ps2init που αρχικοποιεί την συσκευή PS2 όπως αναλύσαμε στη βιβλιοθήκη ps2.c.Τέλος, καλεί την ρουτίνα putchr για να στείλει τον χαρακτήρα '!' ως ένδειξη ότι η αρχικοποιήσει έχει τελειώσει. Μετά ενεργοποιούνται τα interrupts και ορίζονται οι μεταβλητές της ρουτίνας. Τέλος το πρόγραμμα μπαίνει σε ένα βρόγχο for. Σε αυτό τον βρόγχο γίνετε ο υπολογισμός του σήματος ελέγχου και από τον οποίο το πρόγραμμα δεν μπορεί να βγει ποτέ, το μόνο που μπορεί να αλλάξει την ροή του προγράμματος είναι κάποιο interrupt.

Στο πρόγραμμα υπάρχουν τρία ενεργά interrupts, το USART_RXC_vect, το TIMER2_COMP_vect, και το INT0_vect. Τα δυο πρώτα ενεργοποιούνται, το μεν πρώτο, όταν ολοκληρώνεται η λήψη κάποιου byte από την συριακή, και το δεύτερο είναι το interrupt του timer2 και χρησιμοποιείται για τον έλεγχο του Led. Και τα δυο χρησιμοποιούνται για την ανάπτυξη του κώδικα και για debugging και δεν επηρεάζουν την βασική λειτουργιά του προγράμματος που είναι ο έλεγχος του hovercraft. Το τρίτο είναι το εξωτερικό interrupt που συνδέεται με το ποντίκι και ενεργοποιείται όταν αυτό προσπαθεί να στείλει δεδομένα στον επεξεργαστή. Έτσι το ποντίκι είναι αυτό που στέλνοντας δεδομένα θα βγάλει το πρόγραμμα από την for εντολή προκειμένου να γίνουν οι υπόλοιπες λειτουργίες.

Αφού λοιπών κληθεί το interrupt (INT0_vect), αρχικά διαβάζει τα τρία bytes που στέλνει η συσκευή και προσθέτει τα δεδομένα σε δυο integer καταχωρητές x και y που αντιστοιχούν στην κάθετη και οριζόντια μετατόπιση. Θυμίζω εδώ ότι η συσκευή κατάδειξης στέλνει τα δεδομένα σε τρία bytes, αλλά τα δεδομένα αποτελούνται από 9bits για κάθε διάσταση, 8 data bits και ένα bit κατεύθυνσης ή πρόσημο. Στη συνέχεια αυξάνει δυο μετρητές countm και countservo ο πρώτος είναι για τον υπολογισμό του σήματος ελέγχου, και ο δεύτερος για την δημιουργία του σήματος του servo. Το σήμα για το servo δημιουργείται ως έξης. Ο counter ελέγχεται, αμέσως μετά στον κώδικα του interrupt, έτσι ώστε να δημιουργείται το σήμα κάθε πέντε φορές που θα κληθεί το interrupt. Αυτό, σημαίνει ότι για την συχνότητα δειγματοληψίας που έχουμε επιλέξει για το ποντίκι 200samples/sec ή περίοδο 5ms η

περίοδος ανανέωσης του σήματος του servo διαμορφώνεται στα 25ms. Ο χρόνος που θα διαρκέσει το σήμα ορίζεται από τον καταχωρητή U και είναι από 1000-2000 μs. Η καθυστέρηση αυτή δημιουργείται με delay loops, μέσα στο interrupt, αλλά δεν υπάρχει κίνδυνος να γίνει δεύτερη κλήση πριν ολοκληρωθεί η πρώτη διότι ο κώδικας του interrupt εκτελείται σε περίπου 1ms συν 2ms η μεγαλύτερη πιθανή καθυστέρηση για το servo είναι αρκετά λιγότερος χρόνος από τα 5ms που πρέπει να περάσουν πριν η συσκευή στείλει καινούρια δεδομένα. Ο καταχωρητής U αναπαριστά το σήμα ελέγχου καθώς ανάλογα με την τιμή του 0-200 μεταβάλλεται ο χρόνος του σήματος που δίνεται στο servo, μεταξύ της μέγιστης και της ελάχιστης τιμής του 1000-2000μs. Εννοείται βέβαια ότι αν ο U δεν έχει υπολογιστεί ακούμε χρησιμοποιείται η αρχική τιμή του που αντιστοιχεί στην θέση ισορροπίας.

Ο πρώτος μετρητής countm ελέγχεται αφού το πρόγραμμα βγει από το interrupt, μέσα στο for loop. Και κάθε είκοσι κλήσεις του εξωτερικού interrupt, γίνεται ο υπολογισμός του σήματος ελέγχου. Αυτό δίνει συχνότητα υπολογισμού του σήματος περίπου 10Hz. Το interrupt μπορεί να ξανακληθεί κατά την διάρκεια του υπολογισμού του σήματος ελέγχου, αλλά αυτό δεν αποτελεί πρόβλημα καθώς τα δεδομένα έχουν αποθηκευθεί σε άλλες μεταβλητές.

Ας δούμε πως γίνεται ο υπολογισμός του σήματος ελέγχου. Κάθε είκοσι κλίσεις του εξωτερικού interrupt λαμβάνει χώρα η παρακάτω διαδικασία. Πρώτα ανάλογα με την κατάσταση λειτουργίας: κίνηση σε ευθεία, στροφή προς τα αριστερά, ή στροφή προς τα δεξιά, ορίζεται το set point σε ±90° (±π/2) Ή 0°. Μετά ανανεώνονται οι μετρητές που ορίζουν την κατάσταση λειτουργίας και ελέγχουν το led. Μέσω των μεταβλητών mode και countmode το σύστημα εναλλάσσεται μεταξύ των τριών καταστάσεων κάθε μερικά δευτερόλεπτα. Το led ανάβει για να σηματοδοτήσει την αλλαγή κατάστασης.

Στην συνέχεια υπολογίζεται η γωνιά απόκλισης της πορείας του hovercraft από τις δυο μεταβλητές κάθετης και οριζόντιας μετακίνησης xi, και yi, Η γωνία αυτή αντιστοιχεί στην έξοδο του ελεγχόμενου συστήματος, και αφού αφαιρεθεί από το set point έχουμε το σφάλμα. Μετά ελέγχουμε μήπως η γωνία του σφάλματος είναι εκτός των ορίων –π έως π. Και στην συνεχεία γίνεται η υλοποίηση του ελεητή PID. Το αναλογικό κομμάτι γίνεται με απλό πολλαπλασιασμό. Για το διαφορικό κρατάμε στον καταχωρητή Ed το προηγούμενο σφάλμα και το αφαιρούμε από το τωρινό. Και για τον βρόγχο ολοκλήρωσης προσθέτουμε κάθε φορά το σφάλμα στον καταχωρητή Ei.

Ακολούθως υλοποιείται ο ελεγκτής PID. Το αποτέλεσμα αφού περιοριστεί στις μέγιστες τιμές του, θα αποθηκευθεί στον καταχωρητή U ώστε να χρησιμοποιηθεί στην ενεργοποίηση του Servo, και έτσι να κλείσει ο κύκλος του ελέγχου.

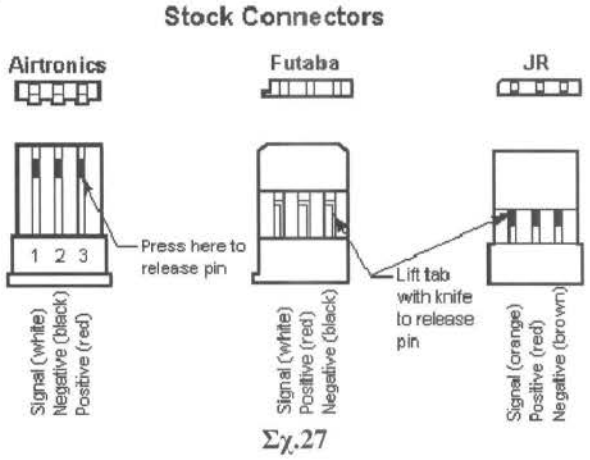
Λειτουργία του SERVO

Σε αυτό σημείο θα περιγράψουμε την λειτουργία του servo, που όπως έχουμε αναφέρει έχει τον ρόλο του μηχανικού ενεργοποιητή στο σύστημα ελέγχου. Αφού δηλαδή έχουμε υπολογίσει την κατάλληλη τιμή της μεταβλητής ελέγχου την τροφοδοτούμε στο servo, το οποίο μετατρέπει το ηλεκτρικό σήμα σε περιστροφή των πτερυγίων κατεύθυνσης. Έχουμε δει στην περιγραφή του μηχανικού συστήματος, ότι η γωνία απόκλισης των πτερυγίων από την θέση "μηδέν" είναι ίση με την απόκλιση του άξονα του servo. Έτσι αρκεί να δώσομε στον άξονα του servo την επιθυμητή απόκλιση.

Το υλικό που χρησιμοποιούμε το έχουμε πάρει από τον χώρο του μοντελισμού. Στην γραμματικότητα οι συσκευές αυτές δεν εκτελούν την λειτουργία ενός servo αλλά κάνουν έναν απλό έλεγχο θέσης. Περιέχουν ένα μικρό μοτέρ με ένα μειωτήρα, και ένα ποτενσιόμετρο για την ανάδραση. Αν το σήμα που δέχονται ανησυχεί σε γωνία διαφορετική από αυτή που βρίσκεται ο άξονας δίνουν την κατάλληλη τάση στο μοτέρ ώστε να μηδενιστεί το σφάλμα.

Στην αγορά υπάρχουν servo που χρησιμοποιούν ψηφιακό κύκλωμα για πετύχουν πιο ακριβή και γρήγορο έλεγχο. Μια συσκευή αυτής της κατηγορίας επιλέχθηκε για την συγκεκριμένη εφαρμογή.

Ας δούμε λοιπόν πως υλοποιείται η επικοινωνία μεταξύ του μικροεπεξεργαστή και του servo. Οι συσκευές αυτές έχουν όπως φαίνεται στο σχήμα 27 τρεις ακροδέκτες:



Σχ.27

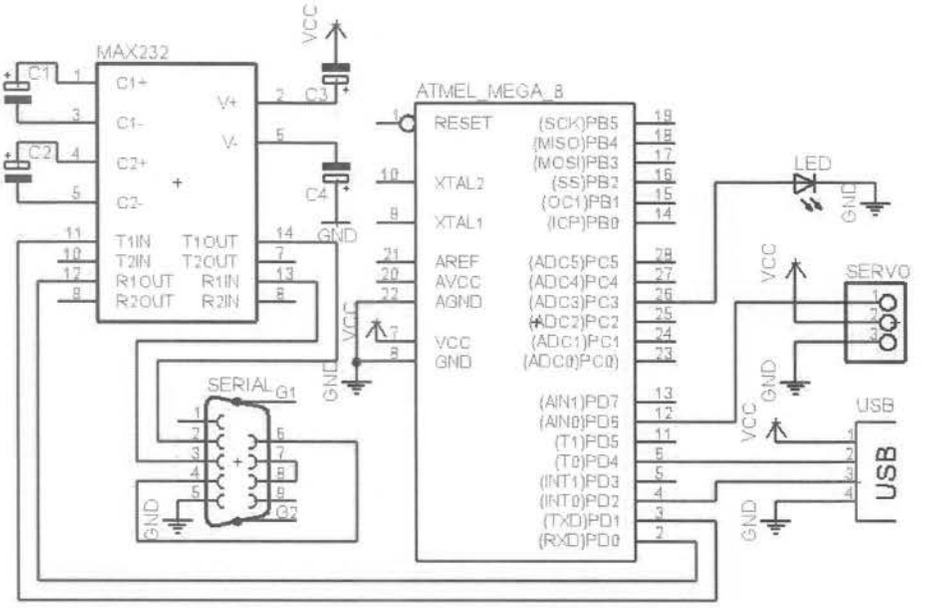
Θετικό, αρνητικό και το σήμα. Η τάση είναι 0 - +5V δηλαδή υπάρχει συμβατότητα με τον μικροεπεξεργαστή και μπορεί να χρησιμοποιηθεί κοινή τροφοδοσία. Η αντίσταση εισόδου στην επαφή του σήματος είναι αρκετά μεγάλη ώστε το servo να οδηγείται κατευθείαν από έναν ακροδέκτη του μικροεπεξεργαστή χωρίς να χρειάζεται άλλο υλικό.

Τέλος το σήμα που δέχονται αυτές οι συσκευές είναι ένας παλμός που ανανεώνεται κάθε 20-30ms η διάρκεια που μένει το σήμα σε υψηλό δυναμικό καθορίζει την επιθυμητή γωνία αναφοράς του servo, και κυμαίνεται από 1000μs έως 2000μs για τα δυο άκρα και η θέση μηδέν βρίσκεται περίπου στα 1500μs. Βλέπουμε ότι η χρονική διαφορά ανάμεσα στο μέγιστο και το ελάχιστο σήμα είναι μόλις 1ms ενώ η περίοδος του σήματος είναι πολύ μεγαλύτερη 20-30ms. Έτσι αν

χρησιμοποιούσαμε έναν 8bit timer για να δημιουργήσουμε ένα σήμα PWM η ανάλυση που θα είχαμε τελικά θα ήταν πολύ μικρή 15-20 δυνατές διαφορετικές θέσεις. 16bit timer όπως έχω αναφέρει δεν υπάρχει διαθέσιμος διότι δεσμεύεται από την σειριακή επικοινωνία, έτσι ως πιο απλή λύση επιλέχθηκε να χρησιμοποιηθούν delay loops αφού πρώτα εξασφαλίστηκε ότι αυτές δεν θα διαταράσσουν της υπόλοιπες λειτουργίες του μικροεπεξεργαστή.

Το ηλεκτρικό κύκλωμα

Αφότου έχουμε αναφερθεί σε όλα τα ηλεκτρικά-ηλεκτρονικά μέρη της κατασκευής, Θα δούμε τώρα και πως έγινε η διασύνδεση μεταξύ τους. Έχω αναφέρει και παραπάνω ότι δεν χρησιμοποίησα κύκλωμα ισχύος τόσο για λογούς απλότητας αλλά και για λόγους ανάδειξης της ποιότητας του ελέγχου. Έτσι οι δυο κινητήρες συνδέονται απλά με έναν διακόπτη. Οπότε ουσιαστικά το κύκλωμα αποτελείται από την σύνδεση του μικροεπεξεργαστή με το περιφερειακό υλικό. Συσκευή κατάδειξης, σειριακή θήρα, servo και ένα led, όπως φαίνεται στο παρακάτω κύκλωμα.

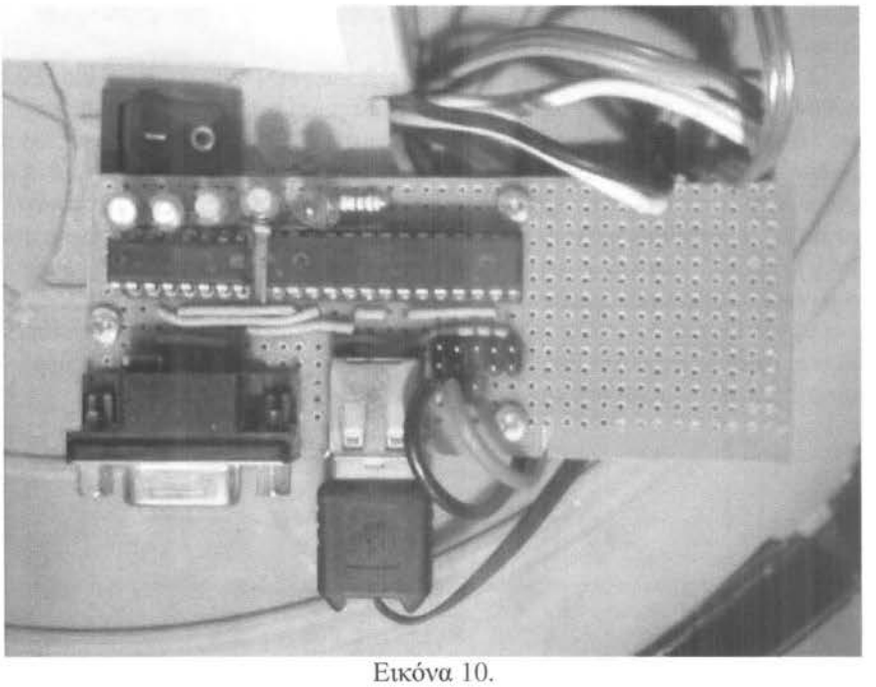


Σχ.28

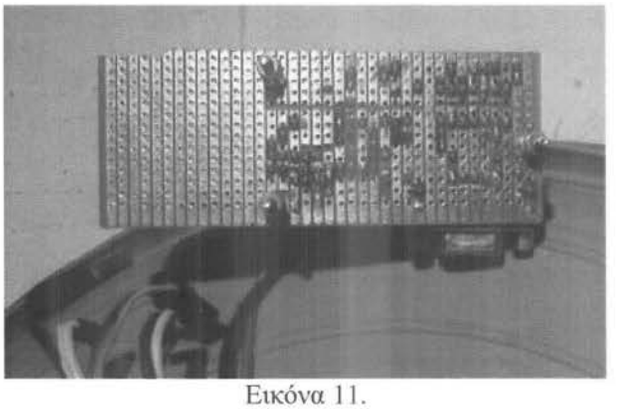
Στο σχήμα βλέπουμε ότι χρησιμοποιούμε μια θύρα USB. Σε αυτή την θύρα συνδέεται η συσκευή κατάδειξης. Έχουμε πει ότι η συσκευή είναι διπλού πρωτοκόλλου PS2- USB. Ο ακροδέκτης της συσκευής είναι USB και μπορεί να συνδεθεί σε PS2 θύρες με την χρήση ενός μετατροπέα. Ο μετατροπέας αυτός δεν έχει καμιά άλλη λειτουργία πέρα από την απ'ευθείας σύνδεση των κατάλληλων ακροδεκτών. Έτσι προκειμένου να αποφύγουμε την χρήση του για λόγους όγκου και ευχρηστίας, αρκεί να κάνουμε τις συνδέσεις έτσι ώστε να αντιστοιχούν στο χρησιμοποιούμενο πρωτόκολλο, PS2. Η αντιστοιχία των συνδέσεων φαίνεται στο παρακάτω σχήμα.

Σχ.29

Σε σχέση με την υλοποίηση του κοιλώματος, δεν έγινε τύπωμα πλακέτας αλλά χρησιμοποιήθηκε πλακέτα με έτοιμες τρύπες και παράλληλους αγωγούς, λόγω της απλότητας του κυκλώματος. Οι συνδέσεις γίνανε στο χέρι και το αποτέλεσμα φαίνεται στις παρακάτω φωτογραφίες.



Εικόνα 10.



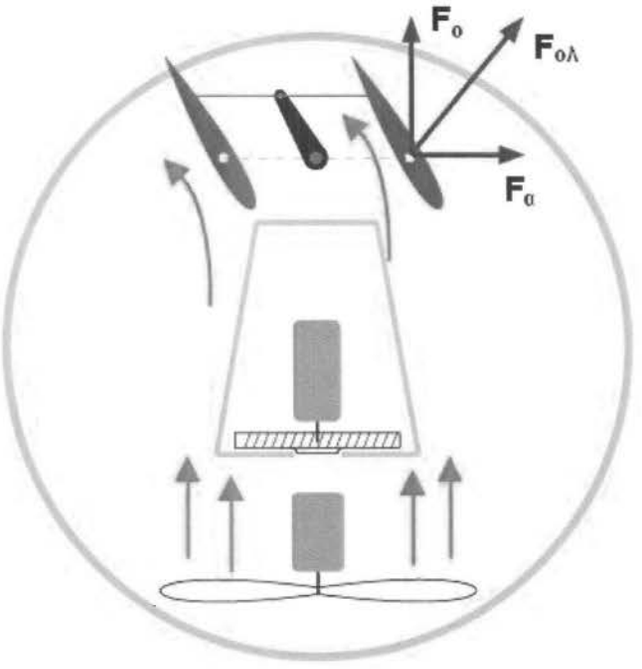
Εικόνα 11.

Το Σύστημα Αυτόματου Ελέγχου

Σε αυτό το κεφάλαιο θα ασχοληθούμε με το σύστημα αυτόματου ελέγχου. Έχουμε αναφερθεί στην πορεία στο ρόλο που έχουν τα επιμέρους στοιχεία της κατασκευής. Ας πιο συγκεκριμένα πως λειτουργούν όλα μαζί.

Έχουμε αναφέρει προηγούμενα, ότι στο hovercraft η κατεύθυνση της ταχύτητας, και ο προσανατολισμός του οχήματος δεν είναι ταυτόσημα. Για την ακρίβεια υπό την προϋπόθεση ότι οι τριβές με το έδαφος είναι πολύ μικρές έως μηδενικές, δεν υπάρχει καμιά άλλη δύναμη που να επιδρά και έτσι η κατεύθυνση της ταχύτητας, και ο προσανατολισμός του οχήματος είναι τελείως ανεξάρτητα. Στην εργασία επικεντρωνόμαστε στον έλεγχο της γωνίας που σχηματίζεται μεταξύ των δυο μεταβλητών και να δούμε αν αυτό μπορεί δώσει αποτελέσματα που θα μπορούν να εντάξουν το σύστημα σε ένα συνολικότερο σύστημα ελέγχου της πορείας και ενδεχομένως και της θέσης του οχήματος.

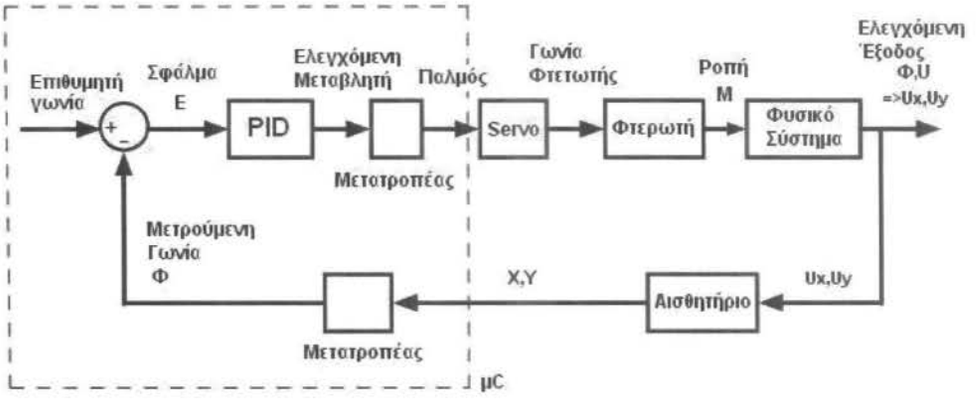
Στόχος μας λοιπόν είναι ο έλεγχος της γωνίας που σχηματίζεται μεταξύ της κατεύθυνσης της ταχύτητας και του άξονα συμμετρίας της κατασκευής. Έχουμε πει ακόμα ότι η εκτροπή της διεύθυνσης της ταχύτητας είναι ο έμμεσος στόχος και δεν μπορούμε να την επηρεάσουμε άμεσα. Αυτό που μπορούμε να επηρεάσουμε πιο άμεσα είναι η διεύθυνση του οχήματος. Για να συμβεί αυτό πρέπει η κατασκευή να περιστραφεί. Προκείμενου να περιστραφεί πρέπει να ασκηθεί ροπή στρέψης, βλέπε ΠΑΡΑΡΤΗΜΑ Ε. Και την ροπή αυτή να μπορούμε να την ελέγχουμε. Έχουμε πει ότι αυτό γίνεται από τα πτερύγια εκτροπής πίσω από την έλικα πρόωσης. Ας δούμε πως γίνεται αυτό. Τα πτερύγιο βρίσκονται μέσα στο ρεύμα αέρα που δημιουργείται από την έλικα. όταν τα πτερύγια βρίσκονται υπό γωνία, σε σχέση με το ρεύμα αέρα δημιουργείται μια πλάγια δύναμη (σχ30).



Σχ.30

Η δύναμη αυτή μπορεί να αναλυθεί σε δυο συνιστώσες την άνωση και την οπισθέλκουσα, βλέπε ΠΑΡΑΡΤΗΜΑ Ε. Η ανάλυση βέβαια των δυο δυνάμεων και της σχέσης μεταξύ τους είναι εκτός του πλαισίου αυτής της εργασίας. Από την ανάλυση όμως της δύναμης στης δυο συνιστώσες, βλέπουμε ότι η κάθετη συνιστώσα δημιουργεί μια ροπή στρέψης στην κατασκευή καθώς ο άξονας που συνδέει το σημείο που ασκείται, με το κέντρο βάρους της κατασκευής είναι κάθετος στην διεύθυνση της δύναμης. Αυτή είναι η ροπή που μας επιτρέπει να ελέγξομε την περιστροφή και κατ'επέκταση την γωνία που περιγράψαμε. Η παράλληλη συνιστώσα περνάει από το κέντρο βάρους και δεν δίνει κάποιο ροπή. Αφαιρείται απλά από την δύναμη πρόωσης που ασκείται στην έλικα και μειώνει την επίδραση της. Δεν επηρεάζει όμως την περιστροφή που εμείς προσπαθούμε να ελέγξουμε.

Ας δούμε λοιπόν πως όλα τα επιμέρους στοιχειά που έχουμε περιγράψει στα προηγούμενα κεφάλαια αλληλεπιδρούν, και πως επιτυγχάνεται η ανάδραση για να έχουμε τελικά ένα κλειστό σύστημα αυτόματου ελέγχου (σχ31).



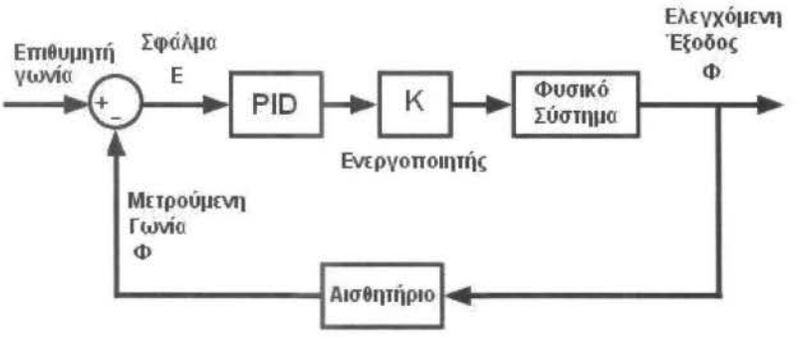
Σχ.31

Ας ξεκινήσουμε την περιγραφή από το σήμα ελέγχου. Το σήμα ελέγχου τροφοδοτείται στον ενεργοποιητή (servo) ο οποίος θα μας δώσει ως έξοδο την γωνία απόκλησης των πτερυγίων εκτροπής φ. Τα πτερύγια κάτω από την επίδραση του ρεύματος αέρα προκαλούν την δημιουργία της ροπής στρέψης M. Η ροπή αυτή επιδρά στο σώμα της κατασκευής προκάνοντας περιστροφή και κατά συνέπεια απόκληση της κατεύθυνσης από την ταχύτητα Φ, την γωνία δηλαδή που ελέγχουμε, την ελεγχόμενη έξοδο.

Την γωνία όμως αυτή δεν την μετράει το αισθητήριο μας κατευθείαν. Και δεν μπορεί να την μετρήσει, ούτε μπορεί να οριστεί καν, αν η κατασκευή δεν εκτελεί παράλληλα μεταφορική κίνηση. Η κίνηση αυτή περιγράφεται από την ταχύτητα U. Η ταχύτητα αυτή είναι που μετράται και πιο συγκεκριμένα οι δυο συνιστώσες της σε σχέση με τους άξονες της κατασκευής x και y. Αφού μετρηθούν οι δυο συνιστώσες από το αισθητήριο μας, την συσκευή κατάδειξης, γίνεται η μετατροπή $\Phi=\tan^{-1}(X/Y)$ Και έτσι εξάγεται η τιμή της ελεγχόμενης εξόδου. Στην συνέχεια η γωνία αυτή αφαιρείται από την επιθυμητή γωνία π.χ. μηδέν για ευθεία μετακίνηση, και έτσι προκύπτει το σφάλμα E. Το σφάλμα είναι η είσοδος του ελεγκτή, εδώ ένας PID, που θα δώσει στην έξοδο την επιθυμητή τιμή της ελεγχόμενης μεταβλητής. Τέλος, η

ελεγχόμενη μεταβλητή θα μετατραπεί στο κατάλληλο παλμό που απαιτεί ο ενεργοποιητής. Έτσι κλείνει ο βρόγχος ελέγχου. Στο σχήμα, μέσα στη διακεκομμένη γραμμή φαίνονται οι λειτουργίες που εκτελεί ο μικροεπεξεργαστής.

Βλέπουμε ότι το σύστημά μας είναι αρκετά περίπλοκο και περιλαμβάνει αρκετά στοιχεία. Σκοπός αυτής της εργασίας δεν είναι ακριβής μαθηματική περιγραφή του. Αλλά μπορούμε εδώ να αναφέρουμε κάποια σημεία επιγραμματικά. Ας προσπαθήσουμε αρχικά να το απλοποιήσουμε λίγο το σύστημά μας. Αρχικά μπορούμε να θεωρήσουμε ότι το αισθητήριο μας διαβάζει κατευθείαν την ελεγχόμενη έξοδο Φ. Αυτό δεν επηρεάζει το σύστημα με την διαφορά ότι αν θέλαμε να είμαστε ακριβείς θα έπρεπε να περιλάβουμε στην ανάλυση μια μικρή χρονική καθυστέρηση. Εδώ θα θεωρήσουμε ότι αυτό δεν μας επηρεάζει. Μετά θα αφαιρέσουμε τις ψηφιακές μετατροπές που ο σκοπός τους είναι απλά να μετατρέψουν τα δεδομένα σε μια μορφή κατανοητή από την επόμενη βαθμίδα και δεν έχουν κάποια δυναμική απόκριση. Ακόμα αν υποθέσουμε ότι ο ενεργοποιητής έχει πολύ γρήγορη απόκριση μπορούμε να αντικαταστήσουμε την βαθμίδα με ένα συντελεστή κέρδους Κ. Τέλος σε σχέση με το σύστημα τον πτερύγιων εκτροπής, είπαμε ότι η σχέση που συνδέει την γωνία απόκλισης τους με την δύναμη που αναπτύσσεται, και σε δεύτερο επίπεδο με την ροπή που παράγει η δύναμη αυτή, σίγουρα δεν είναι γραμμική. Μπορούμε να πούμε όμως ότι για μικρή απόκλιση από την θέση μηδέν, μπορεί να προσεγγιστεί από μια γραμμική σχέση Μ=κ*φ. Όπου κ είναι το κέρδος της συγκεκριμένης βαθμίδας. Έτσι μπορούμε να αντικαταστήσουμε το σύστημα του ενεργοποιητή και των πτερύγιων μαζί από μια βαθμίδα που έχει είσοδο το σήμα ελέγχου και έξοδο την ελεγχόμενη μεταβλητή, την ροπή Μ και περιγράφεται από έναν συντελεστή κέρδους Κ. Έτσι καταλήγουμε στο σύστημα του σχήματος (σχ32).



Σχ.32

Βλέπουμε εδώ ότι μετά τις απλοποιήσεις το σύστημα μας είναι ένα απλό σύστημα κλειστού βρόγχου ελέγχου γωνίας με ελεγκτή PID, και θα μπορούσε να μοντελοποιηθεί, κατά προσέγγιση, εάν γνωρίζαμε την ροπή αδράνειας της κατασκευής, το κέρδος του ενεργοποιητή και του αισθητηρίου, και ίσως την χρονική υστέρηση του αισθητηρίου και του ενεργοποιητη. Μια τέτοια ανάλυση δεν είναι στόχος αυτής της εργασίας και δεν θα μπορούσε εύκολα να δώσει αποτελέσματα κάποιας αξίας με τόσες απλοποιήσεις. Παρ' όλα αυτά το σύστημα που εικονίζεται στο τελευταίο σχήμα μας επιτρέπει να βγάλουμε κάποια συμπεράσματα.

Αρχικά ο PID ελεγκτής με σωστή ρύθμιση θα πρέπει να μπορεί να δώσει ικανοποιητικά αποτελέσματα ελέγχου. Αυτό επιβεβαιώθηκε και πειραματικά. Ακόμα, από τα τρία μέρη του ελεγκτή, το αναλογικό (P) και το διαφορικό (D), είναι τα πιο σημαντικά. Καθώς στο ελεγχόμενο σύστημα υπάρχουν ελάχιστες τριβές δεν μπορεί να υπάρξει παραμένον σφάλμα ελέγχου. Έτσι το ολοκληρωτικό κομμάτι έλεγχου, (I) δεν είναι σίγουρα απαραίτητο. Αυτό, επίσης επιβεβαιώθηκε πειραματικά, καθώς σε μια καλή ρύθμιση των παραγόντων P και D η εισαγωγή ολοκληρωτικού μέρους στον ελεγκτή δεν προσέφερε καλύτερα αποτελέσματα. Επίσης αφού δεν έχουμε μαθηματικό μοντέλο η ρύθμιση του ελεγκτή πρέπει να γίνει εμπειρικά. Αυτό δεν αποτελεί μειονέκτημα ως προς την ποιότητα του ελέγχου διότι σε πολλές περιπτώσεις η εμπειρική ρύθμιση μπορεί να βελτιώσει τα αποτελέσματα άλλων μεθόδων ρύθμισης. Αποτελεί όμως μειονέκτημα ως προς το χρόνο και τις δοκιμές που χρειάζεται και στο ότι δεν υπάρχουν αρχικές τιμές αναφοράς για τους συντελεστές του ελεγκτή (Kp, Ki, Kd). Έτσι για αρχικές τιμές χρησιμοποιήσαμε Ki=0, Kd=0, και Kp=1 δηλαδή η γωνία ελέγχου στα πτερύγια εκτροπής φ να είναι περίπου ίση με την το σφάλμα της γωνίας της ελεγχόμενης εξόδου. Στο πρόγραμμα αυτό αντιστοιχεί σε Kp=100.

Συμπεράσματα

Έχουμε πλέον ολοκληρώσει την ανάλυση της εξέλιξης, κατασκευής και λειτουργίας του οχήματος που αποτελεί το θέμα της εργασίας αυτής.

Τα πειραματική του λειτουργία μας έχει δώσει ικανοποιητικά αποτελέσματα όσον αφορά την ποιότητα ελέγχου, τόσο στην κίνηση σε ευθεία όσο και στην στροφή, δεξιά ή αριστερά. Συνοψίζοντας όλα τα παραπάνω μπορούμε να πούμε ότι ο βασικός στόχος της εργασίας έχει ολοκληρωθεί επιτυχώς.

Εδώ θα ήθελα να αναφέρω ότι κεντρικό στοιχείο στην εξέλιξη και την δομή της εργασίας ήταν η επιλογή του αισθητηρίου, η χρήση του οποίου είχε αρκετές δυσκολίες. Αυτές ήταν: Η τοποθέτηση του, και ο ιδιαίτερος τρόπος που χρησιμοποιήθηκε για να λυθεί αυτό το πρόβλημα. Το γεγονός ότι επέβαλε την ανάπτυξη οδηγού για την επικοινωνία με τον μικροεπεξεργαστή. Αυτό ήταν ίσως το πιο δύσκολο κομμάτι της εργασίας, και το έχουμε αναπτύξει εκτενώς στις προηγούμενες σελίδες. Τέλος, οι προϋποθέσεις που απαντιούνται για λειτουργήσει το αισθητήριο σωστά είναι αρκετές, και έτσι απαιτούν τόσο ακρίβια στην κατασκευή, όσο και εισάγουν περιορισμούς, όσο αφορά στην επιφάνεια κίνησης και την μέγιστη ταχύτητα. Από την άλλη όμως, η επιλογή αυτή, επιτρέπει την μέτρηση της μεταβλητής που θέλουμε να ελέγξουμε, με ικανοποιητική ακρίβεια, ταχύτητα, και με έναν απλό μετασχηματισμό. Και έτσι συμβάλει αρκετά στην τελική επιτυχία του όλου εγχρήματος.

Κλείνοντας, θα ήθελα να αναφέρω ότι παρά τους περιορισμούς της, η κατασκευή θα μπορούσε να αποτελέσει βάση για περαιτέρω εξέλιξη, π.χ. Πλήρη μαθηματική ανάλυση, Επέκταση του ελέγχου διεύθυνσης σε έλεγχο πορείας. Ή ακόμα και προσθήκη τηλεχειρισμού για όχι πλήρως αυτόνομη λειτουργία.

ΠΑΡΑΡΤΗΜΑΤΑ

ΠΑΡΑΡΤΗΜΑ Ά

Στο παράρτημα αυτό παραθέτω μέρος από το manual του κατασκευαστή του μικροεπεξεργαστή, όπου περιλαμβάνονται γενικές πληροφορίες και οι λειτουργίες που χρησιμοποιήθηκαν για την εκπόνηση της εργασίας.

Features

- High-performance, Low-power AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 130 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- Nonvolatile Program and Data Memories
  - 8K Bytes of In-System Self-Programmable Flash
    Endurance: 10,000 Write/Erase Cycles
  - Optional Boot Code Section with Independent Lock Bits
    In-System Programming by On-chip Boot Program
    True Read-While-Write Operation
  - 512 Bytes EEPROM
    Endurance: 100,000 Write/Erase Cycles
  - 1K Byte Internal SRAM
  - Programming Lock for Software Security
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler, one Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Three PWM Channels
  - 8-channel ADC in TQFP and MLF package
    Six Channels 10-bit Accuracy
    Two Channels 8-bit Accuracy
  - 6-channel ADC in PDIP package
    Four Channels 10-bit Accuracy
    Two Channels 8-bit Accuracy
  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated RC Oscillator
  - External and Internal Interrupt Sources
  - Five Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, and Standby
- I/O and Packages
  - 23 Programmable I/O Lines
  - 28-lead PDIP, 32-lead TQFP, and 32-pad MLF
- Operating Voltages
  - 2.7 - 5.5V (ATmega8L)
  - 4.5 - 5.5V (ATmega8)
- Speed Grades
  - 0 - 8 MHz (ATmega8L)
  - 0 - 16 MHz (ATmega8)
- Power Consumption at 4 Mhz, 3V, 25°C
  - Active: 3.6 mA
  - Idle Mode: 1.0 mA
  - Power-down Mode: 0.5 µA

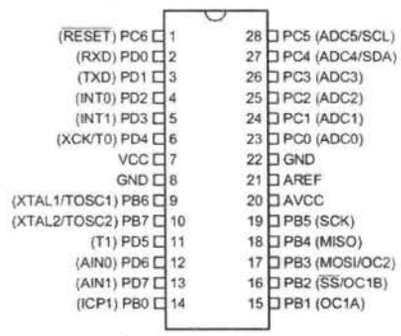
8-bit AVR® with 8K Bytes In-System Programmable Flash

ATmega8
ATmega8L

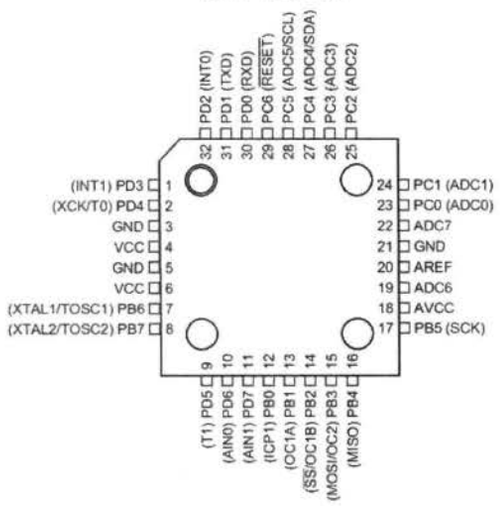
Rev. 2486M–AVR–12/03

Pin Configurations

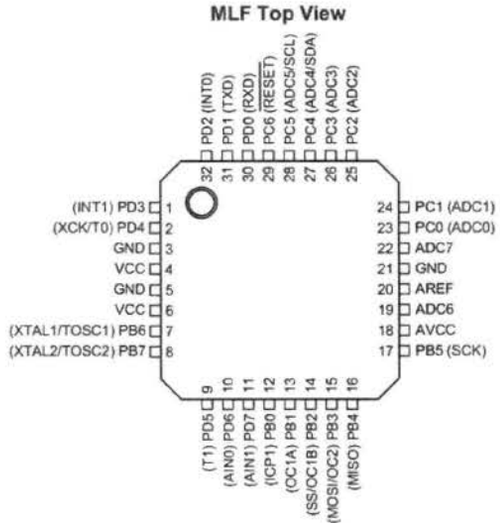
PDIP



TQFP Top View



MLF Top View

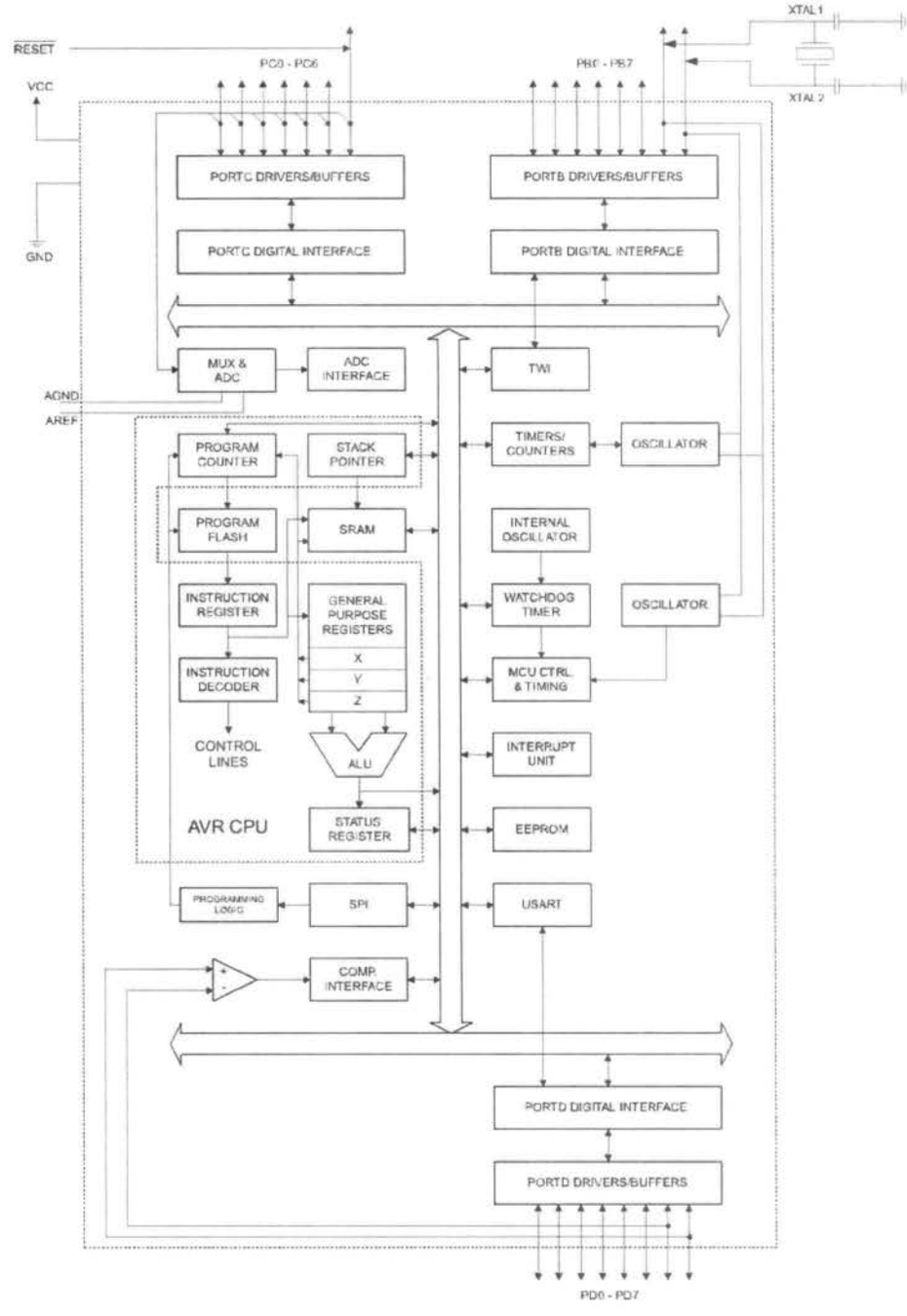


Overview

The ATmega8 is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega8 achieves throughputs approaching 1 MIPS per MHz, allowing the system designer to optimize power consumption versus processing speed.

Block Diagram

Figure 1. Block Diagram



The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATmega8 provides the following features: 8K bytes of In-System Programmable Flash with Read-While-Write capabilities, 512 bytes of EEPROM, 1K byte of SRAM, 23 general purpose I/O lines, 32 general purpose working registers, three flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, a byte oriented Two-wire Serial Interface, a 6-channel ADC (eight channels in TQFP and MLF packages) where four (six) channels have 10-bit accuracy and two channels have 8-bit accuracy, a programmable Watchdog Timer with Internal Oscillator, an SPI serial port, and five software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next Interrupt or Hardware Reset. In Power-save mode, the asynchronous timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except asynchronous timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption.

The device is manufactured using Atmel's high density non-volatile memory technology. The Flash Program memory can be reprogrammed In-System through an SPI serial interface, by a conventional non-volatile memory programmer, or by an On-chip boot program running on the AVR core. The boot program can use any interface to download the application program in the Application Flash memory. Software in the Boot Flash Section will continue to run while the Application Flash Section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega8 is a powerful microcontroller that provides a highly-flexible and cost-effective solution to many embedded control applications.

The ATmega8 AVR is supported with a full suite of program and system development tools, including C compilers, macro assemblers, program debugger/simulators, In-Circuit Emulators, and evaluation kits.

Disclaimer

Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Min and Max values will be available after the device is characterized.

Pin Descriptions

VCC

Digital supply voltage.

GND

Ground.

Port B (PB7..PB0) XTAL1/ XTAL2/TOSC1/TOSC2

Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Depending on the clock selection fuse settings, PB6 can be used as input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

Depending on the clock selection fuse settings, PB7 can be used as output from the inverting Oscillator amplifier.

If the Internal Calibrated RC Oscillator is used as chip clock source, PB7..6 is used as TOSC2..1 input for the Asynchronous Timer/Counter2 if the AS2 bit in ASSR is set.

The various special features of Port B are elaborated in "Alternate Functions of Port B" on page 56 and "System Clock and Clock Options" on page 23.

Port C (PC5..PC0)

Port C is an 7-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running.

PC6/$\overline{\text{RESET}}$

If the RSTDISBL Fuse is programmed, PC6 is used as an I/O pin. Note that the electrical characteristics of PC6 differ from those of the other pins of Port C.

If the RSTDISBL Fuse is unprogrammed, PC6 is used as a Reset input. A low level on this pin for longer than the minimum pulse length will generate a Reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 36. Shorter pulses are not guaranteed to generate a Reset.

The various special features of Port C are elaborated on page 59.

Port D (PD7..PD0)

Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port D also serves the functions of various special features of the ATmega8 as listed on page 61.

$\overline{\text{RESET}}$

Reset input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 36. Shorter pulses are not guaranteed to generate a reset.

AVCC

AVCC is the supply voltage pin for the A/D Converter, Port C (3..0), and ADC (7..6). It should be externally connected to V_{CC}, even if the ADC is not used. If the ADC is used, it should be connected to V_{CC} through a low-pass filter. Note that Port C (5..4) use digital supply voltage, V_{CC}.

AREF

AREF is the analog reference pin for the A/D Converter.

ADC7..6 (TQFP and MLF Package Only)

In the TQFP and MLF package, ADC7..6 serve as analog inputs to the A/D converter. These pins are powered from the analog supply and serve as 10-bit ADC channels.

About Code Examples

This datasheet contains simple code examples that briefly show how to use various parts of the device. These code examples assume that the part specific header file is included before compilation. Be aware that not all C compiler vendors include bit definitions in the header files and interrupt handling in C is compiler dependent. Please confirm with the C compiler documentation for more details.

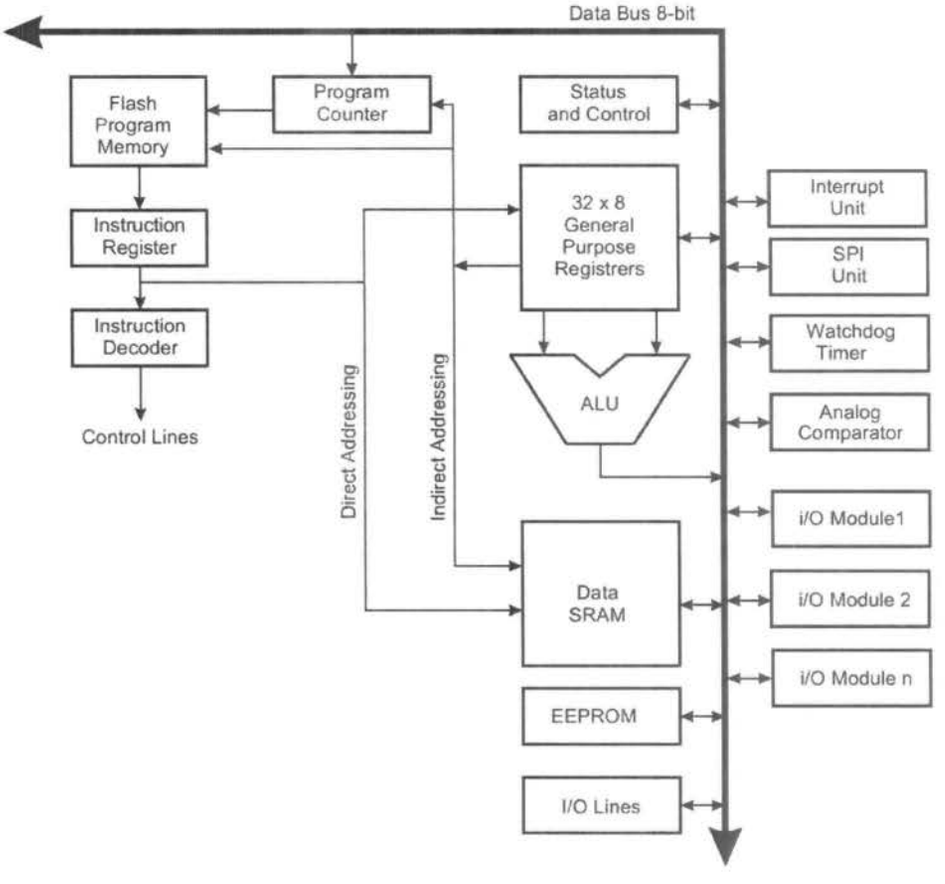
AVR CPU Core

Introduction

This section discusses the AVR core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

Architectural Overview

Figure 2. Block Diagram of the AVR MCU Architecture



In order to maximize performance and parallelism, the AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the Program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the Program memory. This concept enables instructions to be executed in every clock cycle. The Program memory is In-System Reprogrammable Flash memory.

The fast-access Register File contains 32 x 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File – in one clock cycle.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing – enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in Flash Program memory. These added function registers are the 16-bit X-, Y-, and Z-register, described later in this section.

The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the Status Register is updated to reflect information about the result of the operation.

The Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every Program memory address contains a 16- or 32-bit instruction.

Program Flash memory space is divided in two sections, the Boot program section and the Application program section. Both sections have dedicated Lock Bits for write and read/write protection. The SPM instruction that writes into the Application Flash memory section must reside in the Boot program section.

During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine (before subroutines or interrupts are executed). The Stack Pointer SP is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture.

The memory spaces in the AVR architecture are all linear and regular memory maps.

A flexible interrupt module has its control registers in the I/O space with an additional global interrupt enable bit in the Status Register. All interrupts have a separate Interrupt Vector in the Interrupt Vector table. The interrupts have priority in accordance with their Interrupt Vector position. The lower the Interrupt Vector address, the higher the priority.

The I/O memory space contains 64 addresses for CPU peripheral functions as Control Registers, SPI, and other I/O functions. The I/O Memory can be accessed directly, or as the Data Space locations following those of the Register File, 0x20 - 0x5F.

Arithmetic Logic Unit – ALU

The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into three main categories – arithmetic, logical, and bit-functions. Some implementations of the architecture also provide a powerful multiplier supporting both signed/unsigned multiplication and fractional format. See the "Instruction Set" section for a detailed description.

Status Register

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations. Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code.

The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the Instruction Set Reference.

- **Bit 6 – T: Bit Copy Storage**

The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

- **Bit 5 – H: Half Carry Flag**

The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry is useful in BCD arithmetic. See the "Instruction Set Description" for detailed information.

- **Bit 4 – S: Sign Bit, S = N ⊕ V**

The S-bit is always an exclusive or between the Negative Flag N and the Two's Complement Overflow Flag V. See the "Instruction Set Description" for detailed information.

- **Bit 3 – V: Two's Complement Overflow Flag**

The Two's Complement Overflow Flag V supports two's complement arithmetics. See the "Instruction Set Description" for detailed information.

- **Bit 2 – N: Negative Flag**

The Negative Flag N indicates a negative result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 1 – Z: Zero Flag**

The Zero Flag Z indicates a zero result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 0 – C: Carry Flag**

The Carry Flag C indicates a Carry in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

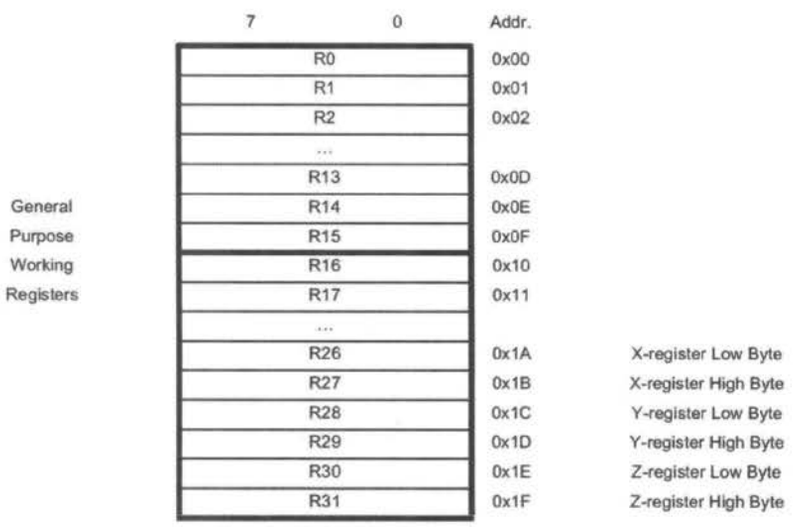
General Purpose Register File

The Register File is optimized for the AVR Enhanced RISC instruction set. In order to achieve the required performance and flexibility, the following input/output schemes are supported by the Register File:

- One 8-bit output operand and one 8-bit result input.
- Two 8-bit output operands and one 8-bit result input.
- Two 8-bit output operands and one 16-bit result input.
- One 16-bit output operand and one 16-bit result input.

Figure 3 shows the structure of the 32 general purpose working registers in the CPU.

Figure 3. AVR CPU General Purpose Working Registers



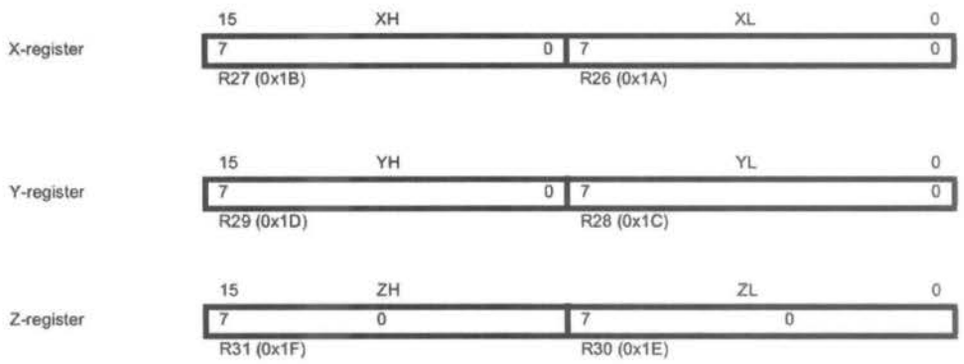
Most of the instructions operating on the Register File have direct access to all registers, and most of them are single cycle instructions.

As shown in Figure 3, each register is also assigned a Data memory address, mapping them directly into the first 32 locations of the user Data Space. Although not being physically implemented as SRAM locations, this memory organization provides great flexibility in access of the registers, as the X-, Y-, and Z-pointer Registers can be set to index any register in the file.

The X-register, Y-register and Z-register

The registers R26..R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the Data Space. The three indirect address registers X, Y and Z are defined as described in Figure 4.

Figure 4. The X-, Y- and Z-Registers



In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement (see the Instruction Set Reference for details).

Stack Pointer

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack PUSH command decreases the Stack Pointer.

The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. This Stack space in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The Stack Pointer must be set to point above 0x60. The Stack Pointer is decremented by one when data is pushed onto the Stack with the PUSH instruction, and it is decremented by two when the return address is pushed onto the Stack with subroutine call or interrupt. The Stack Pointer is incremented by one when data is popped from the Stack with the POP instruction, and it is incremented by two when address is popped from the Stack with return from subroutine RET or return from interrupt RETI.

The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits actually used is implementation dependent. Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Instruction Execution Timing

This section describes the general access timing concepts for instruction execution. The AVR CPU is driven by the CPU clock clk_{CPU}, directly generated from the selected clock source for the chip. No internal clock division is used.

Figure 5 shows the parallel instruction fetches and instruction executions enabled by the Harvard architecture and the fast-access Register File concept. This is the basic pipelining concept to obtain up to 1 MIPS per MHz with the corresponding unique results for functions per cost, functions per clocks, and functions per power-unit.

Figure 5. The Parallel Instruction Fetches and Instruction Executions

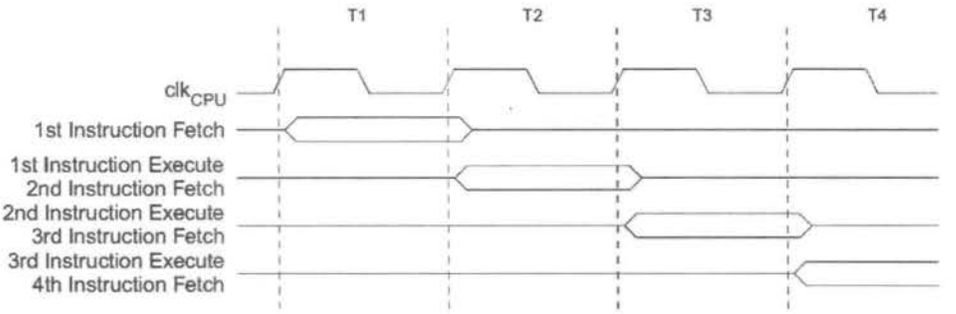
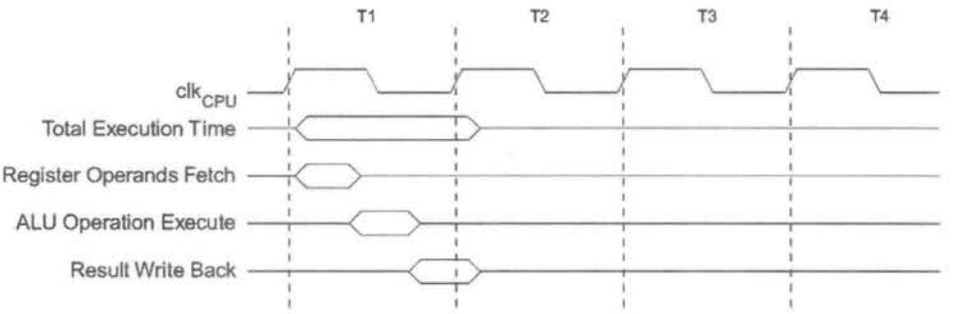


Figure 6 shows the internal timing concept for the Register File. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

Figure 6. Single Cycle ALU Operation



Reset and Interrupt Handling

The AVR provides several different interrupt sources. These interrupts and the separate Reset Vector each have a separate Program Vector in the Program memory space. All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt. Depending on the Program Counter value, interrupts may be automatically disabled when Boot Lock Bits BLB02 or BLB12 are programmed. This feature improves software security. See the section "Memory Programming" on page 219 for details.

The lowest addresses in the Program memory space are by default defined as the Reset and Interrupt Vectors. The complete list of Vectors is shown in "Interrupts" on page 44. The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request 0. The Interrupt Vectors can be moved to the start

of the boot Flash section by setting the Interrupt Vector Select (IVSEL) bit in the General Interrupt Control Register (GICR). Refer to "Interrupts" on page 44 for more information. The Reset Vector can also be moved to the start of the boot Flash section by programming the BOOTRST Fuse, see "Boot Loader Support – Read-While-Write Self-Programming" on page 206.

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.

There are basically two types of interrupts. The first type is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector in order to execute the interrupt handling routine, and hardware clears the corresponding Interrupt Flag. Interrupt Flags can also be cleared by writing a logic one to the flag bit position(s) to be cleared. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software. Similarly, if one or more interrupt conditions occur while the global interrupt enable bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the global interrupt enable bit is set, and will then be executed by order of priority.

The second type of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.

Note that the Status Register is not automatically stored when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software.

When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction. The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

Assembly Code Example

```
in r16, SREG     ; store SREG value
cli    ; disable interrupts during timed sequence
sbi EECR, EEMWE  ; start EEPROM write
sbi EECR, EEWE
out SREG, r16    ; restore SREG value (I-bit)
```

C Code Example

```
char cSREG;
cSREG = SREG; /* store SREG value */
/* disable interrupts during timed sequence */
_CLI();
EECR |= (1<<EEMWE); /* start EEPROM write */
EECR |= (1<<EEWE);
SREG = cSREG; /* restore SREG value (I-bit) */
```

When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, as shown in the following example.

Assembly Code Example

```
sei   ; set global interrupt enable
sleep; enter sleep, waiting for interrupt
; note: will enter sleep before any pending
; interrupt(s)
```

C Code Example

```
_SEI(); /* set global interrupt enable */
_SLEEP(); /* enter sleep, waiting for interrupt */
/* note: will enter sleep before any pending interrupt(s) */
```

Interrupt Response Time

The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum. After four clock cycles, the Program Vector address for the actual interrupt handling routine is executed. During this 4-clock cycle period, the Program Counter is pushed onto the Stack. The Vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served. If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (2 bytes) is popped back from the Stack, the Stack Pointer is incremented by 2, and the I-bit in SREG is set.

AVR ATmega8 Memories

This section describes the different memories in the ATmega8. The AVR architecture has two main memory spaces, the Data memory and the Program Memory space. In addition, the ATmega8 features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

In-System Reprogrammable Flash Program Memory

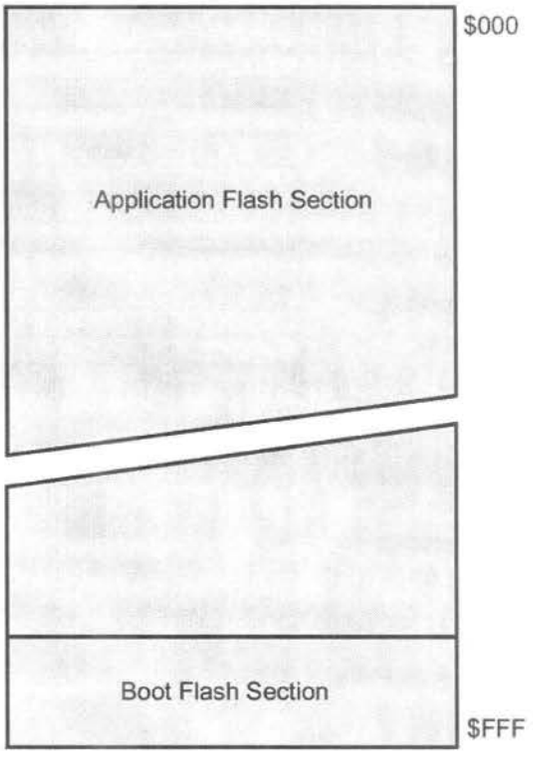
The ATmega8 contains 8K bytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16- or 32-bits wide, the Flash is organized as 4K x 16 bits. For software security, the Flash Program memory space is divided into two sections, Boot Program section and Application Program section.

The Flash memory has an endurance of at least 10,000 write/erase cycles. The ATmega8 Program Counter (PC) is 12 bits wide, thus addressing the 4K Program memory locations. The operation of Boot Program section and associated Boot Lock Bits for software protection are described in detail in "Boot Loader Support – Read-While-Write Self-Programming" on page 206. "Memory Programming" on page 219 contains a detailed description on Flash Programming in SPI- or Parallel Programming mode.

Constant tables can be allocated within the entire Program memory address space (see the LPM – Load Program memory instruction description).

Timing diagrams for instruction fetch and execution are presented in "Instruction Execution Timing" on page 12.

Figure 7. Program Memory Map



SRAM Data Memory

Figure 8 shows how the ATmega8 SRAM Memory is organized.

The lower 1120 Data memory locations address the Register File, the I/O Memory, and the internal data SRAM. The first 96 locations address the Register File and I/O Memory, and the next 1024 locations address the internal data SRAM.

The five different addressing modes for the Data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the Register File, registers R26 to R31 feature the indirect addressing pointer registers.

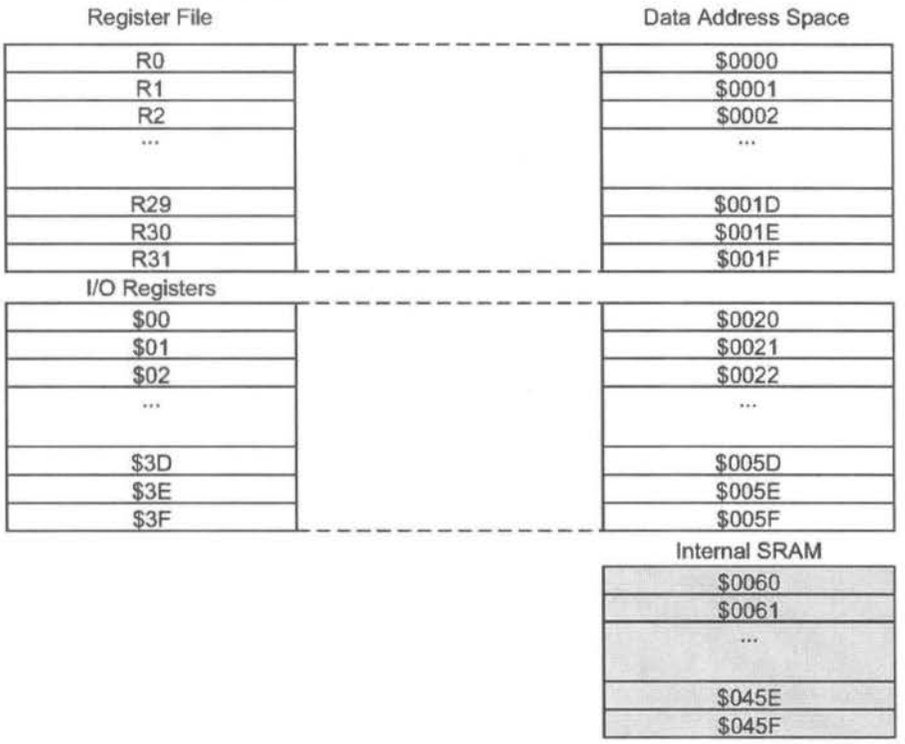
The direct addressing reaches the entire data space.

The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Z-register.

When using register indirect addressing modes with automatic pre-decrement and post-increment, the address registers X, Y and Z are decremented or incremented.

The 32 general purpose working registers, 64 I/O Registers, and the 1024 bytes of internal data SRAM in the ATmega8 are all accessible through all these addressing modes. The Register File is described in "General Purpose Register File" on page 10.

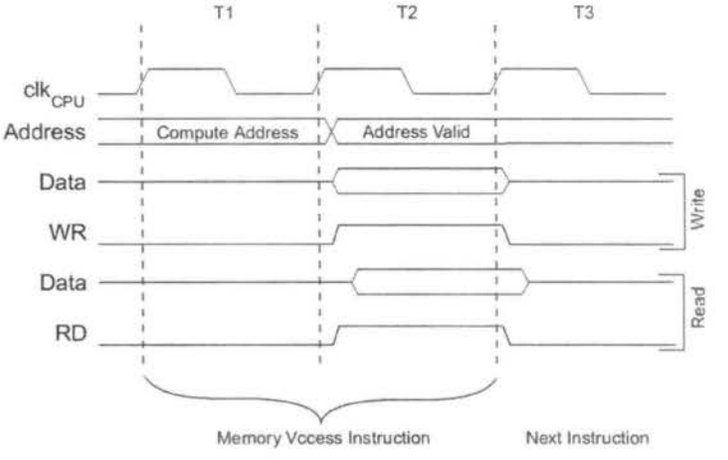
Figure 8. Data Memory Map



Data Memory Access Times

This section describes the general access timing concepts for internal memory access. The internal data SRAM access is performed in two clk_{CPU} cycles as described in Figure 9.

Figure 9. On-chip Data SRAM Access Cycles



EEPROM Data Memory

The ATmega8 contains 512 bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described bellow, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

"Memory Programming" on page 219 contains a detailed description on EEPROM Programming in SPI- or Parallel Programming mode.

EEPROM Read/Write Access

The EEPROM Access Registers are accessible in the I/O space.

The write access time for the EEPROM is given in Table 1 on page 19. A self-timing function, however, lets the user software detect when the next byte can be written. If the user code contains instructions that write the EEPROM, some precautions must be taken. In heavily filtered power supplies, V_{CC} is likely to rise or fall slowly on Power-up/down. This causes the device for some period of time to run at a voltage lower than specified as minimum for the clock frequency used. See "Preventing EEPROM Corruption" on page 21. for details on how to avoid problems in these situations.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to the description of the EEPROM Control Register for details on this.

When the EEPROM is read, the CPU is halted for four clock cycles before the next instruction is executed. When the EEPROM is written, the CPU is halted for two clock cycles before the next instruction is executed.

The EEPROM Address Register – EEARH and EEARL

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	–	EEAR8	EEARH
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

- **Bits 15..9 – Res: Reserved Bits**

These bits are reserved bits in the ATmega8 and will always read as zero.

- **Bits 8..0 – EEAR8..0: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL – specify the EEPROM address in the 512 bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 511. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

The EEPROM Data Register – EEDR

Bit	7	6	5	4	3	2	1	0	
	MSB							LSB	EEDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..0 – EEDR7..0: EEPROM Data**

For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

The EEPROM Control Register – EECR

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	EERIE	EEMWE	EEWE	EERE	EECR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	X	0	

- **Bits 7..4 – Res: Reserved Bits**

These bits are reserved bits in the ATmega8 and will always read as zero.

- **Bit 3 – EERIE: EEPROM Ready Interrupt Enable**

Writing EERIE to one enables the EEPROM Ready Interrupt if the I bit in SREG is set. Writing EERIE to zero disables the interrupt. The EEPROM Ready interrupt generates a constant interrupt when EEWE is cleared.

- **Bit 2 – EEMWE: EEPROM Master Write Enable**

The EEMWE bit determines whether setting EEWE to one causes the EEPROM to be written. When EEMWE is set, setting EEWE within four clock cycles will write data to the EEPROM at the selected address If EEMWE is zero, setting EEWE will have no effect. When EEMWE has been written to one by software, hardware clears the bit to zero after four clock cycles. See the description of the EEWE bit for an EEPROM write procedure.

- **Bit 1 – EEWE: EEPROM Write Enable**

The EEPROM Write Enable Signal EEWE is the write strobe to the EEPROM. When address and data are correctly set up, the EEWE bit must be written to one to write the

value into the EEPROM. The EEMWE bit must be written to one before a logical one is written to EEWE, otherwise no EEPROM write takes place. The following procedure should be followed when writing the EEPROM (the order of steps 3 and 4 is not essential):

1. Wait until EEWE becomes zero.
2. Wait until SPMEN in SPMCR becomes zero.
3. Write new EEPROM address to EEAR (optional).
4. Write new EEPROM data to EEDR (optional).
5. Write a logical one to the EEMWE bit while writing a zero to EEWE in EECR.
6. Within four clock cycles after setting EEMWE, write a logical one to EEWE.

The EEPROM can not be programmed during a CPU write to the Flash memory. The software must check that the Flash programming is completed before initiating a new EEPROM write. Step 2 is only relevant if the software contains a boot loader allowing the CPU to program the Flash. If the Flash is never being updated by the CPU, step 2 can be omitted. See "Boot Loader Support – Read-While-Write Self-Programming" on page 206 for details about boot programming.

Caution: An interrupt between step 5 and step 6 will make the write cycle fail, since the EEPROM Master Write Enable will time-out. If an interrupt routine accessing the EEPROM is interrupting another EEPROM access, the EEAR or EEDR Register will be modified, causing the interrupted EEPROM access to fail. It is recommended to have the Global Interrupt Flag cleared during all the steps to avoid these problems.

When the write access time has elapsed, the EEWE bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEWE has been set, the CPU is halted for two cycles before the next instruction is executed.

- **Bit 0 – EERE: EEPROM Read Enable**

The EEPROM Read Enable Signal EERE is the read strobe to the EEPROM. When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed.

The user should poll the EEWE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR Register.

The calibrated Oscillator is used to time the EEPROM accesses. Table 1 lists the typical programming time for EEPROM access from the CPU.

Table 1. EEPROM Programming Time

Symbol	Number of Calibrated RC Oscillator Cycles[1]	Typ Programming Time
EEPROM Write (from CPU)	8448	8.5 ms

Note: 1. Uses 1 MHz clock, independent of CKSEL Fuse settings.

The following code examples show one assembly and one C function for writing to the EEPROM. The examples assume that interrupts are controlled (for example by disabling interrupts globally) so that no interrupts will occur during execution of these functions. The examples also assume that no Flash boot loader is present in the software. If such code is present, the EEPROM write function must also wait for any ongoing SPM command to finish.

Assembly Code Example

```
EEPROM_write:
  ; Wait for completion of previous write
  sbic EECR,EEWE
  rjmp EEPROM_write
  ; Set up address (r18:r17) in address register
  out EEARH, r18
  out EEARL, r17
  ; Write data (r16) to data register
  out EEDR,r16
  ; Write logical one to EEMWE
  sbi EECR,EEMWE
  ; Start eeprom write by setting EEWE
  sbi EECR,EEWE
  ret
```

C Code Example

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEWE))
    ;
  /* Set up address and data registers */
  EEAR = uiAddress;
  EEDR = ucData;
  /* Write logical one to EEMWE */
  EECR |= (1<<EEMWE);
  /* Start eeprom write by setting EEWE */
  EECR |= (1<<EEWE);
}
```

The next code examples show assembly and C functions for reading the EEPROM. The examples assume that interrupts are controlled so that no interrupts will occur during execution of these functions.

Assembly Code Example

```
EEPROM_read:
  ; Wait for completion of previous write
  sbic EECR,EEWE
  rjmp EEPROM_read
  ; Set up address (r18:r17) in address register
  out EEARH, r18
  out EEARL, r17
  ; Start eeprom read by writing EERE
  sbi EECR,EERE
  ; Read data from data register
  in  r16,EEDR
  ret
```

C Code Example

```
unsigned char EEPROM_read(unsigned int uiAddress)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEWE))
    ;
  /* Set up address register */
  EEAR = uiAddress;
  /* Start eeprom read by writing EERE */
  EECR |= (1<<EERE);
  /* Return data from data register */
  return EEDR;
}
```

EEPROM Write during Power-down Sleep Mode

When entering Power-down sleep mode while an EEPROM write operation is active, the EEPROM write operation will continue, and will complete before the Write Access time has passed. However, when the write operation is completed, the Oscillator continues running, and as a consequence, the device does not enter Power-down entirely. It is therefore recommended to verify that the EEPROM write operation is completed before entering Power-down.

Preventing EEPROM Corruption

During periods of low V_{CC}, the EEPROM data can be corrupted because the supply voltage is too low for the CPU and the EEPROM to operate properly. These issues are the same as for board level systems using EEPROM, and the same design solutions should be applied.

An EEPROM data corruption can be caused by two situations when the voltage is too low. First, a regular write sequence to the EEPROM requires a minimum voltage to operate correctly. Second, the CPU itself can execute instructions incorrectly, if the supply voltage is too low.

EEPROM data corruption can easily be avoided by following this design recommendation:

Keep the AVR RESET active (low) during periods of insufficient power supply voltage. This can be done by enabling the internal Brown-out Detector (BOD). If the detection level of the internal BOD does not match the needed detection level, an external low V_{CC} Reset Protection circuit can be used. If a reset occurs while a write operation is in progress, the write operation will be completed provided that the power supply voltage is sufficient.

I/O Memory

The I/O space definition of the ATmega8 is shown in "" on page 282.

All ATmega8 I/Os and peripherals are placed in the I/O space. The I/O locations are accessed by the IN and OUT instructions, transferring data between the 32 general purpose working registers and the I/O space. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions. Refer to the instruction set section for more details. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses.

For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.

Some of the Status Flags are cleared by writing a logical one to them. Note that the CBI and SBI instructions will operate on all bits in the I/O Register, writing a one back into any flag read as set, thus clearing the flag. The CBI and SBI instructions work with registers 0x00 to 0x1F only.

The I/O and Peripherals Control Registers are explained in later sections.

Interrupts

This section describes the specifics of the interrupt handling performed by the ATmega8. For a general explanation of the AVR interrupt handling, refer to "Reset and Interrupt Handling" on page 12.

Interrupt Vectors in ATmega8

Table 18. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	0x000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	TIMER2 COMP	Timer/Counter2 Compare Match
5	0x004	TIMER2 OVF	Timer/Counter2 Overflow
6	0x005	TIMER1 CAPT	Timer/Counter1 Capture Event
7	0x006	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	0x007	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	0x008	TIMER1 OVF	Timer/Counter1 Overflow
10	0x009	TIMER0 OVF	Timer/Counter0 Overflow
11	0x00A	SPI, STC	Serial Transfer Complete
12	0x00B	USART, RXC	USART, Rx Complete
13	0x00C	USART, UDRE	USART Data Register Empty
14	0x00D	USART, TXC	USART, Tx Complete
15	0x00E	ADC	ADC Conversion Complete
16	0x00F	EE_RDY	EEPROM Ready
17	0x010	ANA_COMP	Analog Comparator
18	0x011	TWI	Two-wire Serial Interface
19	0x012	SPM_RDY	Store Program Memory Ready

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 206.

2. When the IVSEL bit in GICR is set, Interrupt Vectors will be moved to the start of the boot Flash section. The address of each Interrupt Vector will then be the address in this table added to the start address of the boot Flash section.

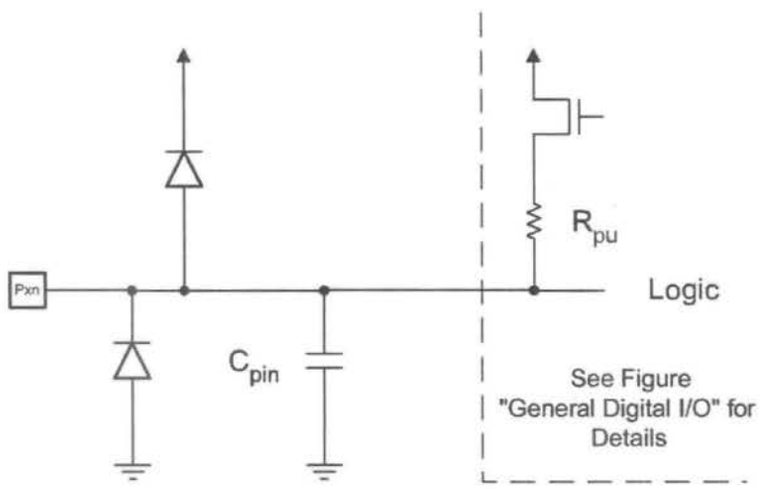
Table 19 shows reset and Interrupt Vectors placement for the various combinations of BOOTRST and IVSEL settings. If the program never enables an interrupt source, the Interrupt Vectors are not used, and regular program code can be placed at these locations. This is also the case if the Reset Vector is in the Application section while the Interrupt Vectors are in the boot section or vice versa.

I/O Ports

Introduction

All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the SBI and CBI instructions. The same applies when changing drive value (if configured as output) or enabling/disabling of pull-up resistors (if configured as input). Each output buffer has symmetrical drive characteristics with both high sink and source capability. The pin driver is strong enough to drive LED displays directly. All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance. All I/O pins have protection diodes to both V_{CC} and Ground as indicated in Figure 21. Refer to "Electrical Characteristics" on page 237 for a complete list of parameters.

Figure 21. I/O Pin Equivalent Schematic



All registers and bit references in this section are written in general form. A lower case "x" represents the numbering letter for the port, and a lower case "n" represents the bit number. However, when using the register or bit defines in a program, the precise form must be used (i.e., PORTB3 for bit 3 in Port B, here documented generally as PORTxn). The physical I/O Registers and bit locations are listed in "Register Description for I/O Ports" on page 63.

Three I/O memory address locations are allocated for each port, one each for the Data Register – PORTx, Data Direction Register – DDRx, and the Port Input Pins – PINx. The Port Input Pins I/O location is read only, while the Data Register and the Data Direction Register are read/write. In addition, the Pull-up Disable – PUD bit in SFIOR disables the pull-up function for all pins in all ports when set.

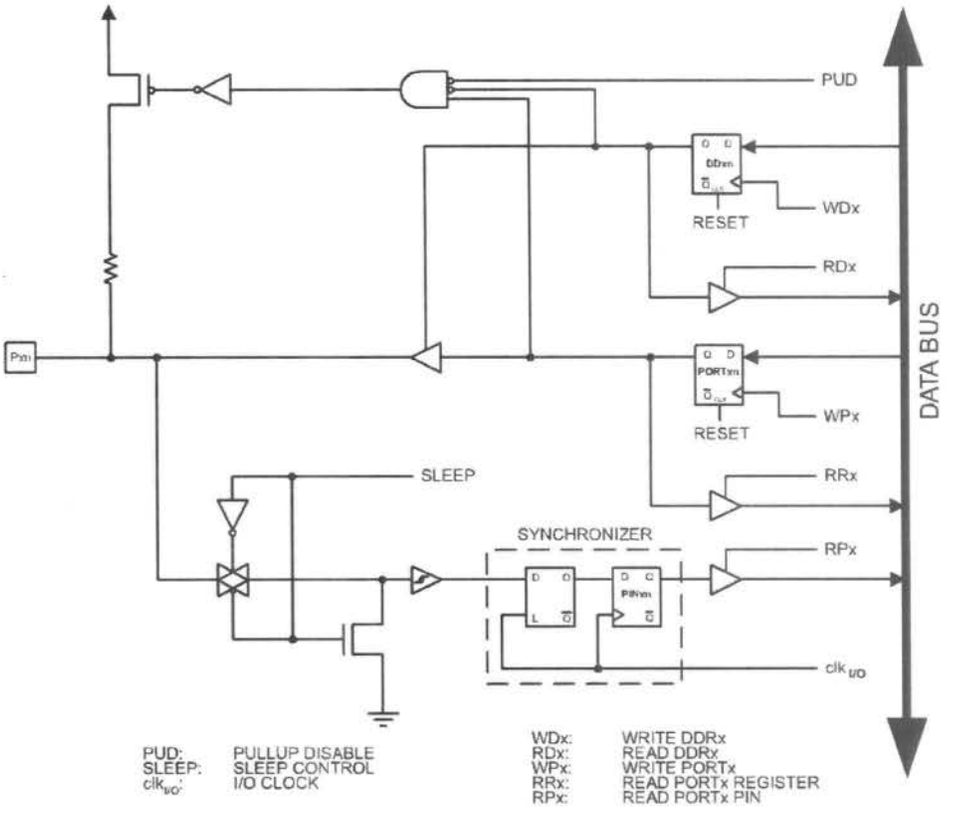
Using the I/O port as General Digital I/O is described in "Ports as General Digital I/O" on page 50. Most port pins are multiplexed with alternate functions for the peripheral features on the device. How each alternate function interferes with the port pin is described in "Alternate Port Functions" on page 54. Refer to the individual module sections for a full description of the alternate functions.

Note that enabling the alternate function of some of the port pins does not affect the use of the other pins in the port as general digital I/O.

Ports as General Digital I/O

The ports are bi-directional I/O ports with optional internal pull-ups. Figure 22 shows a functional description of one I/O port pin, here generically called Pxn.

Figure 22. General Digital I/O[1]



Note: 1. WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

Configuring the Pin

Each port pin consists of 3 Register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description for I/O Ports" on page 63, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when a reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

When switching between tri-state ({DDxn, PORTxn} = 0b00) and output high ({DDxn, PORTxn} = 0b11), an intermediate state with either pull-up enabled ({DDxn, PORTxn} = 0b01) or output low ({DDxn, PORTxn} = 0b10) must occur. Normally, the pull-up enabled state is fully acceptable, as a high-impedant environment will not notice the difference between a strong high driver and a pull-up. If this is not the case, the PUD bit in the SFIOR Register can be set to disable all pull-ups in all ports.

Switching between input with pull-up and output low generates the same problem. The user must use either the tri-state ({DDxn, PORTxn} = 0b00) or the output high state ({DDxn, PORTxn} = 0b11) as an intermediate step.

Table 20 summarizes the control signals for the pin value.

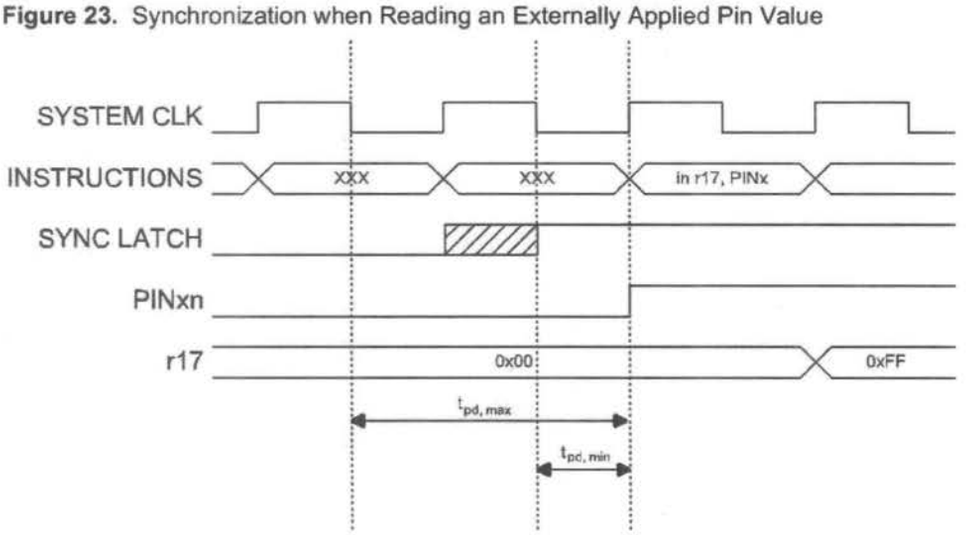
Table 20. Port Pin Configurations

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if external pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Reading the Pin Value

Independent of the setting of Data Direction bit DDxn, the port pin can be read through the PINxn Register Bit. As shown in Figure 22, the PINxn Register bit and the preceding latch constitute a synchronizer. This is needed to avoid metastability if the physical pin changes value near the edge of the internal clock, but it also introduces a delay. Figure 23 shows a timing diagram of the synchronization when reading an externally applied pin value. The maximum and minimum propagation delays are denoted $t_{pd,max}$ and $t_{pd,min}$, respectively.

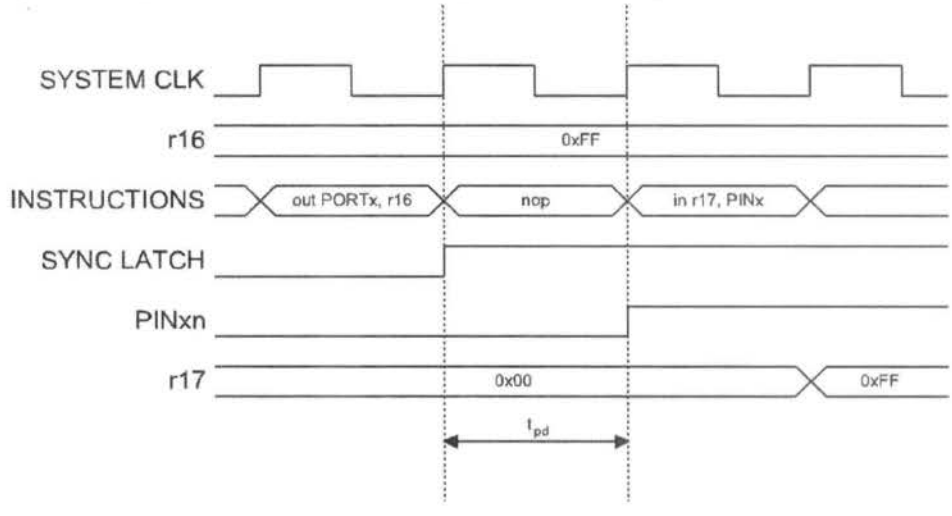
Figure 23. Synchronization when Reading an Externally Applied Pin Value



Consider the clock period starting shortly *after* the first falling edge of the system clock. The latch is closed when the clock is low, and goes transparent when the clock is high, as indicated by the shaded region of the "SYNC LATCH" signal. The signal value is latched when the system clock goes low. It is clocked into the PINxn Register at the succeeding positive clock edge. As indicated by the two arrows $t_{pd,max}$ and $t_{pd,min}$, a single signal transition on the pin will be delayed between ½ and 1-½ system clock period depending upon the time of assertion.

When reading back a software assigned pin value, a *nop* instruction must be inserted as indicated in Figure 24. The *out* instruction sets the "SYNC LATCH" signal at the positive edge of the clock. In this case, the delay t_{pd} through the synchronizer is 1 system clock period.

Figure 24. Synchronization when Reading a Software Assigned Pin Value



The following code example shows how to set port B pins 0 and 1 high, 2 and 3 low, and define the port pins from 4 to 7 as input with pull-ups assigned to port pins 6 and 7. The resulting pin values are read back again, but as previously discussed, a *nop* instruction is included to be able to read back the value recently assigned to some of the pins.

Assembly Code Example[1]

```
...
; Define pull-ups and set outputs high
; Define directions for port pins
ldi  r16,(1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0)
ldi  r17,(1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0)
out  PORTB,r16
out  DDRB,r17
; Insert nop for synchronization
nop
; Read port pins
in   r16,PINB
...
```

C Code Example[1]

```
unsigned char i;
...
/* Define pull-ups and set outputs high */
/* Define directions for port pins */
PORTB = (1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0);
DDRB = (1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
/* Insert nop for synchronization*/
_NOP();
/* Read port pins */
i = PINB;
...
```

Note: 1. For the assembly program, two temporary registers are used to minimize the time from pull-ups are set on pins 0, 1, 6, and 7, until the direction bits are correctly set, defining bit 2 and 3 as low and redefining bits 0 and 1 as strong high drivers.

Digital Input Enable and Sleep Modes

As shown in Figure 22, the digital input signal can be clamped to ground at the input of the Schmitt-trigger. The signal denoted SLEEP in the figure, is set by the MCU Sleep Controller in Power-down mode, Power-save mode, and Standby mode to avoid high power consumption if some input signals are left floating, or have an analog signal level close to $V_{CC}/2$.

SLEEP is overridden for port pins enabled as External Interrupt pins. If the External Interrupt Request is not enabled, SLEEP is active also for these pins. SLEEP is also overridden by various other alternate functions as described in "Alternate Port Functions" on page 54.

If a logic high level ("one") is present on an Asynchronous External Interrupt pin configured as "Interrupt on Rising Edge, Falling Edge, or Any Logic Change on Pin" while the external interrupt is *not* enabled, the corresponding External Interrupt Flag will be set when resuming from the above mentioned sleep modes, as the clamping in these sleep modes produces the requested logic change.

Unconnected pins

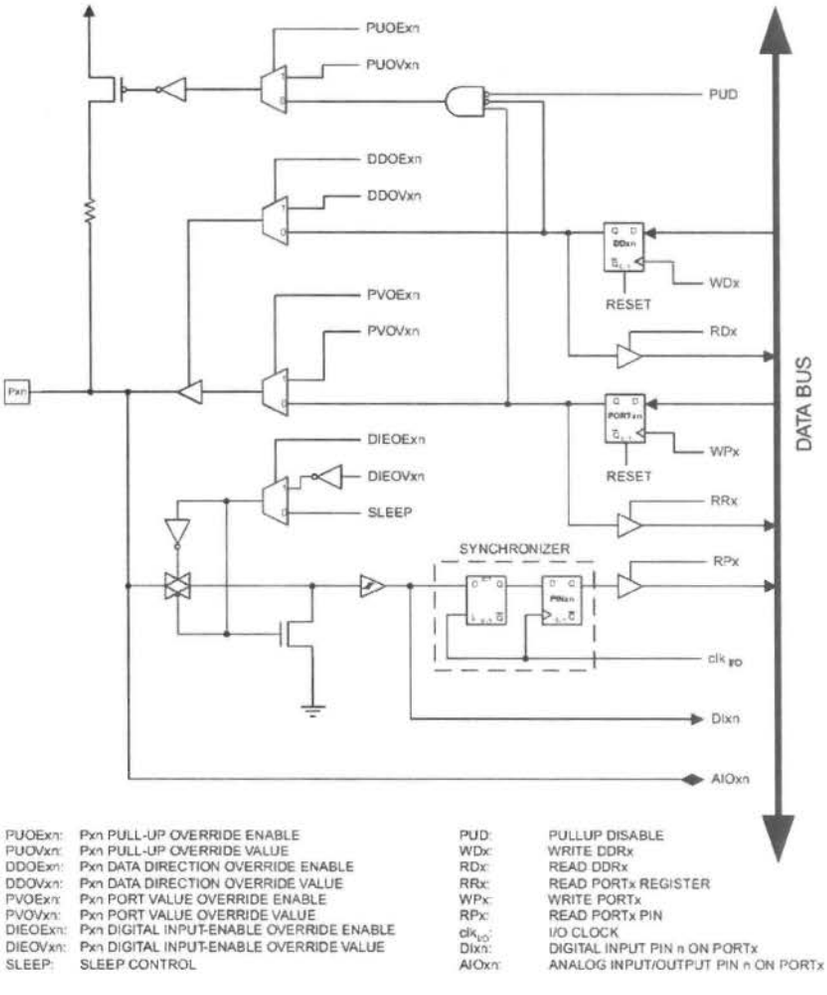
If some pins are unused, it is recommended to ensure that these pins have a defined level. Even though most of the digital inputs are disabled in the deep sleep modes as described above, floating inputs should be avoided to reduce current consumption in all other modes where the digital inputs are enabled (Reset, Active mode and Idle mode).

The simplest method to ensure a defined level of an unused pin, is to enable the internal pull-up. In this case, the pull-up will be disabled during reset. If low power consumption during reset is important, it is recommended to use an external pull-up or pull-down. Connecting unused pins directly to V_{CC} or GND is not recommended, since this may cause excessive currents if the pin is accidentally configured as an output.

Alternate Port Functions

Most port pins have alternate functions in addition to being general digital I/Os. Figure 25 shows how the port pin control signals from the simplified Figure 22 can be overridden by alternate functions. The overriding signals may not be present in all port pins, but the figure serves as a generic description applicable to all port pins in the AVR microcontroller family.

Figure 25. Alternate Port Functions[1]



PUOExn:	Pxn PULL-UP OVERRIDE ENABLE	PUD:	PULLUP DISABLE
PUOVxn:	Pxn PULL-UP OVERRIDE VALUE	WDx:	WRITE DDRx
DDOExn:	Pxn DATA DIRECTION OVERRIDE ENABLE	RDx:	READ DDRx
DDOVxn:	Pxn DATA DIRECTION OVERRIDE VALUE	RRx:	READ PORTx REGISTER
PVOExn:	Pxn PORT VALUE OVERRIDE ENABLE	WPx:	WRITE PORTx
PVOVxn:	Pxn PORT VALUE OVERRIDE VALUE	RPx:	READ PORTx PIN
DIEOExn:	Pxn DIGITAL INPUT-ENABLE OVERRIDE ENABLE	$clk_{I/O}$:	I/O CLOCK
DIEOVxn:	Pxn DIGITAL INPUT-ENABLE OVERRIDE VALUE	DIxn:	DIGITAL INPUT PIN n ON PORTx
SLEEP:	SLEEP CONTROL	AIOxn:	ANALOG INPUT/OUTPUT PIN n ON PORTx

Note: 1. WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports. All other signals are unique for each pin.

Register Description for I/O Ports

The Port B Data Register – PORTB

Bit	7	6	5	4	3	2	1	0	
	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port B Data Direction Register – DDRB

Bit	7	6	5	4	3	2	1	0	
	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port B Input Pins Address – PINB

Bit	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The Port C Data Register – PORTC

Bit	7	6	5	4	3	2	1	0	
	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORTC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port C Data Direction Register – DDRC

Bit	7	6	5	4	3	2	1	0	
	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port C Input Pins Address – PINC

Bit	7	6	5	4	3	2	1	0	
	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	PINC
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The Port D Data Register – PORTD

Bit	7	6	5	4	3	2	1	0	
	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port D Data Direction Register – DDRD

Bit	7	6	5	4	3	2	1	0	
	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port D Input Pins Address – PIND

Bit	7	6	5	4	3	2	1	0	
	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

External Interrupts

The external interrupts are triggered by the INT0, and INT1 pins. Observe that, if enabled, the interrupts will trigger even if the INT0..1 pins are configured as outputs. This feature provides a way of generating a software interrupt. The external interrupts can be triggered by a falling or rising edge or a low level. This is set up as indicated in the specification for the MCU Control Register – MCUCR. When the external interrupt is enabled and is configured as level triggered, the interrupt will trigger as long as the pin is held low. Note that recognition of falling or rising edge interrupts on INT0 and INT1 requires the presence of an I/O clock, described in "Clock Systems and their Distribution" on page 23. Low level interrupts on INT0/INT1 are detected asynchronously. This implies that these interrupts can be used for waking the part also from sleep modes other than Idle mode. The I/O clock is halted in all sleep modes except Idle mode.

Note that if a level triggered interrupt is used for wake-up from Power-down mode, the changed level must be held for some time to wake up the MCU. This makes the MCU less sensitive to noise. The changed level is sampled twice by the Watchdog Oscillator clock. The period of the Watchdog Oscillator is 1 µs (nominal) at 5.0V and 25°C. The frequency of the Watchdog Oscillator is voltage dependent as shown in "Electrical Characteristics" on page 237. The MCU will wake up if the input has the required level during this sampling or if it is held until the end of the start-up time. The start-up time is defined by the SUT Fuses as described in "System Clock and Clock Options" on page 23. If the level is sampled twice by the Watchdog Oscillator clock but disappears before the end of the start-up time, the MCU will still wake up, but no interrupt will be generated. The required level must be held long enough for the MCU to complete the wake up to trigger the level interrupt.

MCU Control Register – MCUCR

The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 3, 2 – ISC11, ISC10: Interrupt Sense Control 1 Bit 1 and Bit 0**

The External Interrupt 1 is activated by the external pin INT1 if the SREG I-bit and the corresponding interrupt mask in the GICR are set. The level and edges on the external INT1 pin that activate the interrupt are defined in Table 31. The value on the INT1 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

Table 31. Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

- **Bit 1, 0 – ISC01, ISC00: Interrupt Sense Control 0 Bit 1 and Bit 0**

The External Interrupt 0 is activated by the external pin INT0 if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INT0 pin that activate the interrupt are defined in Table 32. The value on the INT0 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

Table 32. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	–	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – INT1: External Interrupt Request 1 Enable**

When the INT1 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control1 bits 1/0 (ISC11 and ISC10) in the MCU general Control Register (MCUCR) define whether the external interrupt is activated on rising and/or falling edge of the INT1 pin or level sensed. Activity on the pin will cause an interrupt request even if INT1 is configured as an output. The corresponding interrupt of External Interrupt Request 1 is executed from the INT1 Interrupt Vector.

- **Bit 6 – INT0: External Interrupt Request 0 Enable**

When the INT0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control0 bits 1/0 (ISC01 and ISC00) in the MCU general Control Register (MCUCR) define whether the external interrupt is activated on rising and/or falling edge of the INT0 pin or level sensed. Activity on the pin will cause an interrupt request even if INT0 is configured as an output. The corresponding interrupt of External Interrupt Request 0 is executed from the INT0 Interrupt Vector.

General Interrupt Flag Register – GIFR

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	–	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – INTF1: External Interrupt Flag 1**

When an event on the INT1 pin triggers an interrupt request, INTF1 becomes set (one). If the I-bit in SREG and the INT1 bit in GICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when INT1 is configured as a level interrupt.

- **Bit 6 – INTF0: External Interrupt Flag 0**

When an event on the INT0 pin triggers an interrupt request, INTF0 becomes set (one). If the I-bit in SREG and the INT0 bit in GICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when INT0 is configured as a level interrupt.

8-bit Timer/Counter0

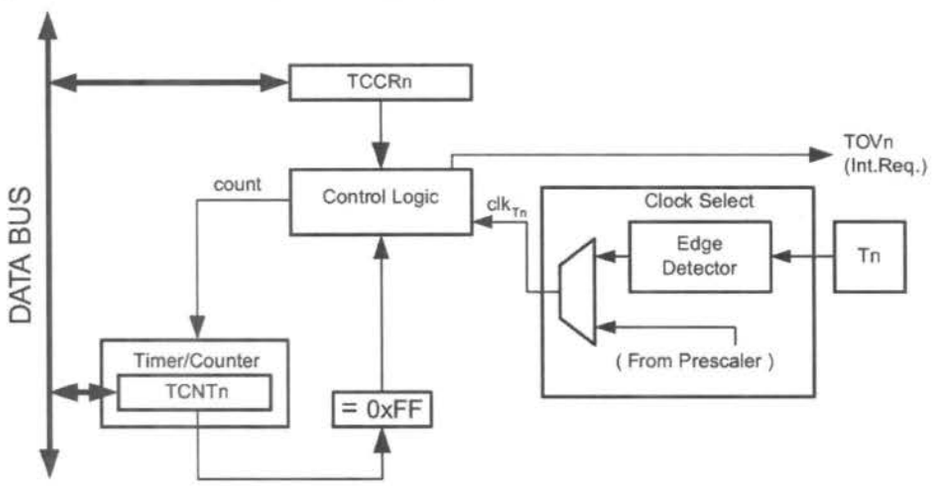
Timer/Counter0 is a general purpose, single channel, 8-bit Timer/Counter module. The main features are:

- **Single Channel Counter**
- **Frequency Generator**
- **External Event Counter**
- **10-bit Clock Prescaler**

Overview

A simplified block diagram of the 8-bit Timer/Counter is shown in Figure 26. For the actual placement of I/O pins, refer to "Pin Configurations" on page 2. CPU accessible I/O Registers, including I/O bits and I/O pins, are shown in bold. The device-specific I/O Register and bit locations are listed in the "8-bit Timer/Counter Register Description" on page 70.

Figure 26. 8-bit Timer/Counter Block Diagram



Registers

The Timer/Counter (TCNT0) is an 8-bit register. Interrupt request (abbreviated to Int. Req. in the figure) signals are all visible in the Timer Interrupt Flag Register (TIFR). All interrupts are individually masked with the Timer Interrupt Mask Register (TIMSK). TIFR and TIMSK are not shown in the figure since these registers are shared by other timer units.

The Timer/Counter can be clocked internally or via the prescaler, or by an external clock source on the T0 pin. The Clock Select logic block controls which clock source and edge the Timer/Counter uses to increment its value. The Timer/Counter is inactive when no clock source is selected. The output from the clock select logic is referred to as the timer clock (clk_{T0}).

Definitions

Many register and bit references in this document are written in general form. A lower case "n" replaces the Timer/Counter number, in this case 0. However, when using the register or bit defines in a program, the precise form must be used i.e. TCNT0 for accessing Timer/Counter0 counter value and so on.

The definitions in Table 33 are also used extensively throughout this datasheet.

Table 33. Definitions

BOTTOM	The counter reaches the BOTTOM when it becomes 0x00
MAX	The counter reaches its MAXimum when it becomes 0xFF (decimal 255)

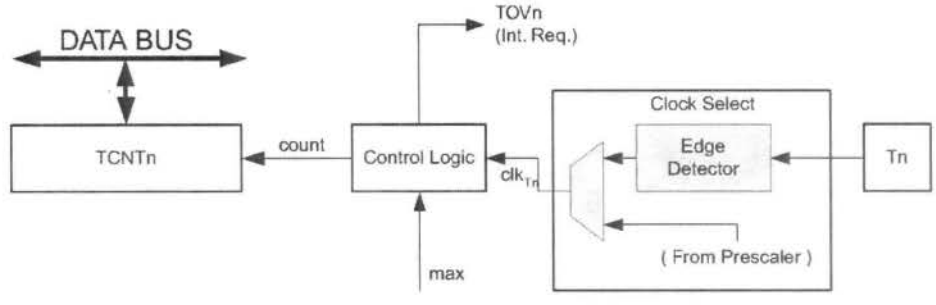
Timer/Counter Clock Sources

The Timer/Counter can be clocked by an internal or an external clock source. The clock source is selected by the clock select logic which is controlled by the clock select (CS02:0) bits located in the Timer/Counter Control Register (TCCR0). For details on clock sources and prescaler, see "Timer/Counter0 and Timer/Counter1 Prescalers" on page 72.

Counter Unit

The main part of the 8-bit Timer/Counter is the programmable counter unit. Figure 27 shows a block diagram of the counter and its surroundings.

Figure 27. Counter Unit Block Diagram



Signal description (internal signals):

count	Increment TCNT0 by 1.
clk$_{Tn}$	Timer/Counter clock, referred to as clk$_{T0}$ in the following.
max	Signalize that TCNT0 has reached maximum value.

The counter is incremented at each timer clock (clk$_{T0}$). clk$_{T0}$ can be generated from an external or internal clock source, selected by the clock select bits (CS02:0). When no clock source is selected (CS02:0 = 0) the timer is stopped. However, the TCNT0 value can be accessed by the CPU, regardless of whether clk$_{T0}$ is present or not. A CPU write overrides (has priority over) all counter clear or count operations.

Operation

The counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (MAX = 0xFF) and then restarts from the bottom (0x00). In normal operation the Timer/Counter Overflow Flag (TOV0) will be set in the same timer clock cycle as the TCNT0 becomes zero. The TOV0 Flag in this case behaves like a ninth bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOV0 Flag, the timer resolution can be increased by software. A new counter value can be written anytime.

Timer/Counter Timing Diagrams

The Timer/Counter is a synchronous design and the timer clock (clk_{T0}) is therefore shown as a clock enable signal in the following figures. The figures include information on when Interrupt Flags are set. Figure 28 contains timing data for basic Timer/Counter operation. The figure shows the count sequence close to the MAX value.

Figure 28. Timer/Counter Timing Diagram, No Prescaling

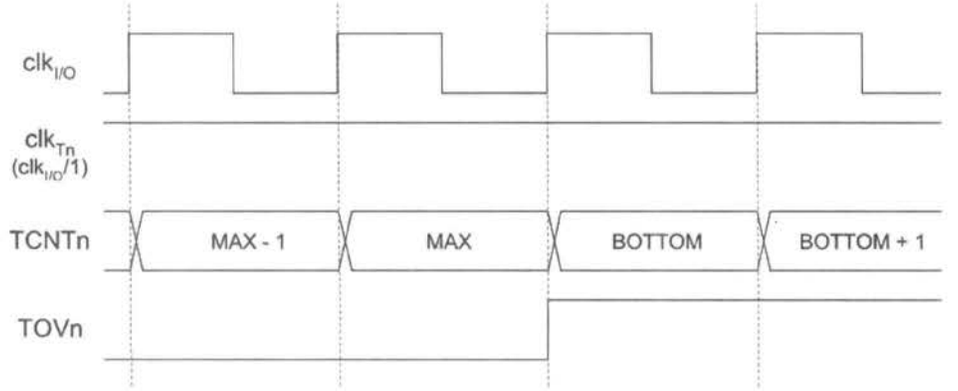
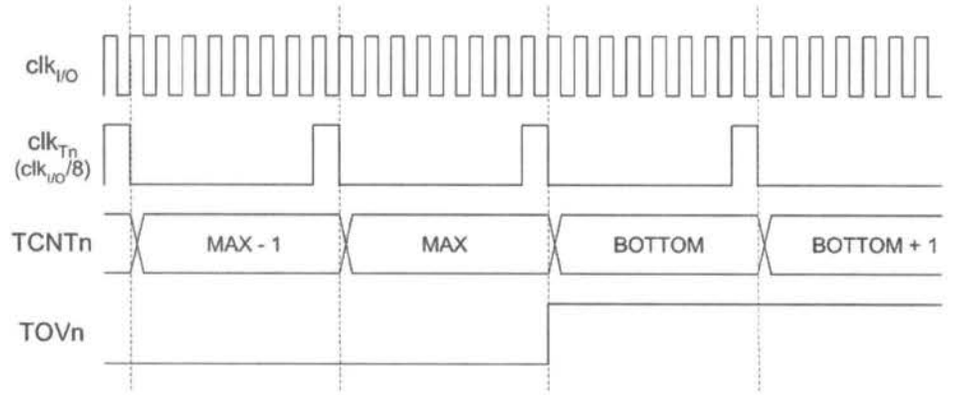


Figure 29 shows the same timing data, but with the prescaler enabled.

Figure 29. Timer/Counter Timing Diagram, with Prescaler ($f_{clk\_I/O}/8$)



8-bit Timer/Counter Register Description

Timer/Counter Control Register – TCCR0

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 2:0 – CS02:0: Clock Select**

The three clock select bits select the clock source to be used by the Timer/Counter.

Table 34. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Timer/Counter Register – TCNT0

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter.

Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

Timer/Counter Interrupt Flag Register – TIFR

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOV0: Timer/Counter0 Overflow Flag**

The bit TOV0 is set (one) when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt Handling Vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE0 (Timer/Counter0 Overflow Interrupt Enable), and TOV0 are set (one), the Timer/Counter0 Overflow interrupt is executed.

USART

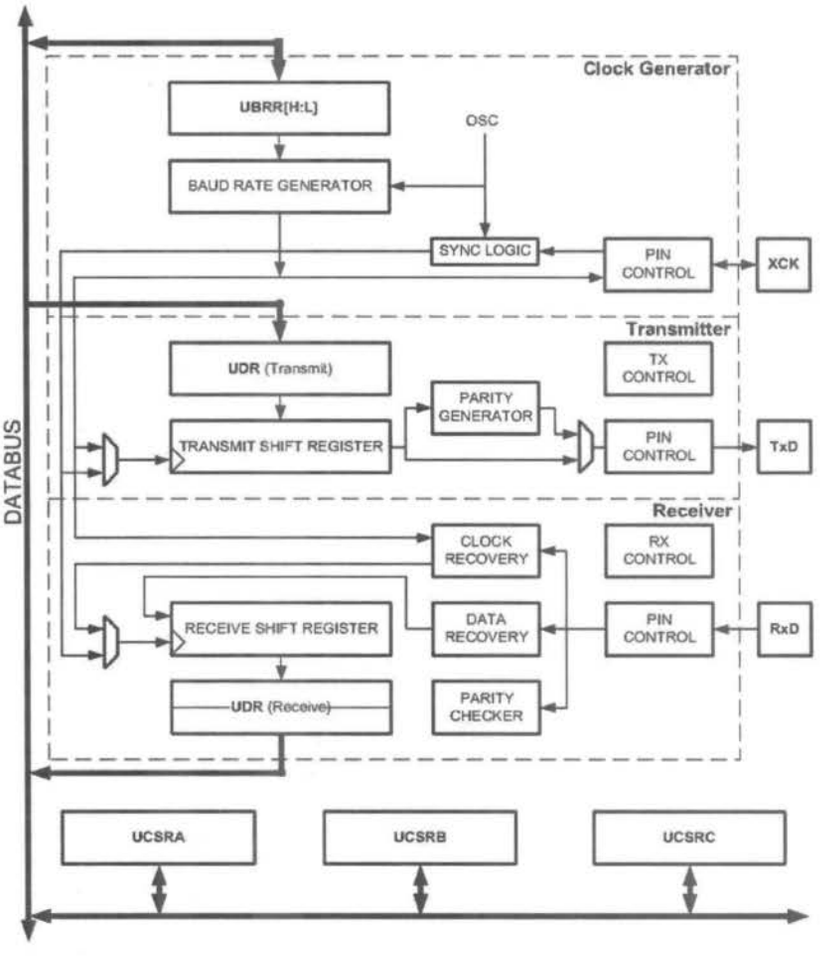
The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly-flexible serial communication device. The main features are:

- **Full Duplex Operation (Independent Serial Receive and Transmit Registers)**
- **Asynchronous or Synchronous Operation**
- **Master or Slave Clocked Synchronous Operation**
- **High Resolution Baud Rate Generator**
- **Supports Serial Frames with 5, 6, 7, 8, or 9 Databits and 1 or 2 Stop Bits**
- **Odd or Even Parity Generation and Parity Check Supported by Hardware**
- **Data OverRun Detection**
- **Framing Error Detection**
- **Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter**
- **Three Separate Interrupts on TX Complete, TX Data Register Empty and RX Complete**
- **Multi-processor Communication Mode**
- **Double Speed Asynchronous Communication Mode**

Overview

A simplified block diagram of the USART Transmitter is shown in Figure 61. CPU accessible I/O Registers and I/O pins are shown in bold.

Figure 61. USART Block Diagram[1]



Note: 1. Refer to "Pin Configurations" on page 2, Table 30 on page 62, and Table 29 on page 62 for USART pin placement.

The dashed boxes in the block diagram separate the three main parts of the USART (listed from the top): Clock generator, Transmitter and Receiver. Control Registers are shared by all units. The clock generation logic consists of synchronization logic for external clock input used by synchronous slave operation, and the baud rate generator. The XCK (transfer clock) pin is only used by synchronous transfer mode. The Transmitter consists of a single write buffer, a serial Shift Register, Parity Generator and control logic for handling different serial frame formats. The write buffer allows a continuous transfer of data without any delay between frames. The Receiver is the most complex part of the USART module due to its clock and data recovery units. The recovery units are used for asynchronous data reception. In addition to the recovery units, the Receiver includes a parity checker, control logic, a Shift Register and a two level receive buffer (UDR). The Receiver supports the same frame formats as the Transmitter, and can detect Frame Error, Data OverRun and Parity Errors.

AVR USART vs. AVR UART – Compatibility

The USART is fully compatible with the AVR UART regarding:

- Bit locations inside all USART Registers.
- Baud Rate Generation.
- Transmitter Operation.
- Transmit Buffer Functionality.
- Receiver Operation.

However, the receive buffering has two improvements that will affect the compatibility in some special cases:

- A second Buffer Register has been added. The two Buffer Registers operate as a circular FIFO buffer. Therefore the UDR must only be read once for each incoming data! More important is the fact that the Error Flags (FE and DOR) and the ninth data bit (RXB8) are buffered with the data in the receive buffer. Therefore the status bits must always be read before the UDR Register is read. Otherwise the error status will be lost since the buffer state is lost.
- The Receiver Shift Register can now act as a third buffer level. This is done by allowing the received data to remain in the serial Shift Register (see Figure 61) if the Buffer Registers are full, until a new start bit is detected. The USART is therefore more resistant to Data OverRun (DOR) error conditions.

The following control bits have changed name, but have same functionality and register location:

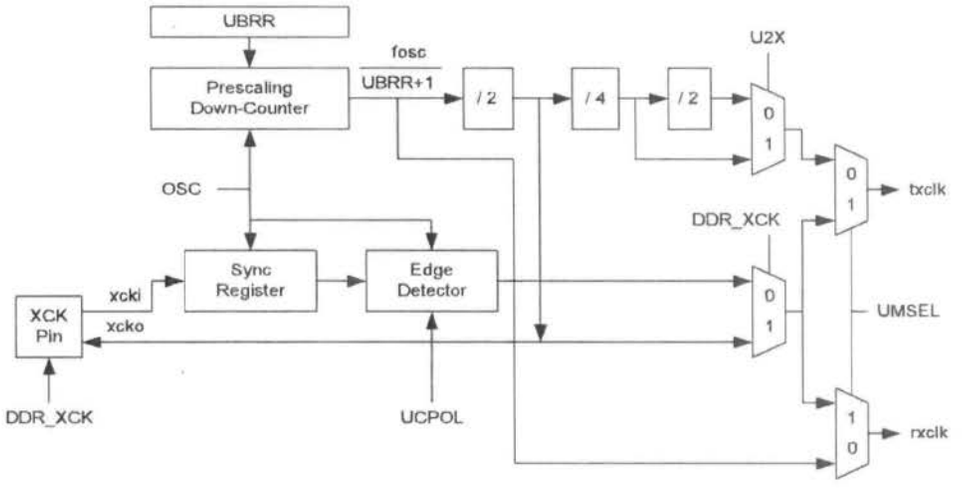
- CHR9 is changed to UCSZ2.
- OR is changed to DOR.

Clock Generation

The clock generation logic generates the base clock for the Transmitter and Receiver. The USART supports four modes of clock operation: normal asynchronous, double speed asynchronous, Master synchronous and Slave Synchronous mode. The UMSEL bit in USART Control and Status Register C (UCSRC) selects between asynchronous and synchronous operation. Double speed (Asynchronous mode only) is controlled by the U2X found in the UCSRA Register. When using Synchronous mode (UMSEL = 1), the Data Direction Register for the XCK pin (DDR_XCK) controls whether the clock source is internal (Master mode) or external (Slave mode). The XCK pin is only active when using Synchronous mode.

Figure 62 shows a block diagram of the clock generation logic.

Figure 62. Clock Generation Logic, Block Diagram



Signal description:

txclk Transmitter clock. (Internal Signal)

rxclk Receiver base clock. (Internal Signal)

xcki Input from XCK pin (internal Signal). Used for synchronous slave operation.

xcko Clock output to XCK pin (Internal Signal). Used for synchronous master operation.

fosc XTAL pin frequency (System Clock).

Internal Clock Generation – The Baud Rate Generator

Internal clock generation is used for the asynchronous and the Synchronous Master modes of operation. The description in this section refers to Figure 62.

The USART Baud Rate Register (UBRR) and the down-counter connected to it function as a programmable prescaler or baud rate generator. The down-counter, running at system clock (fosc), is loaded with the UBRR value each time the counter has counted down to zero or when the UBRRL Register is written. A clock is generated each time the counter reaches zero. This clock is the baud rate generator clock output (= fosc/(UBRR+1)). The Transmitter divides the baud rate generator clock output by 2, 8, or 16 depending on mode. The baud rate generator output is used directly by the Receiver's clock and data recovery units. However, the recovery units use a state machine that uses 2, 8, or 16 states depending on mode set by the state of the UMSEL, U2X and DDR_XCK bits.

Table 52 contains equations for calculating the baud rate (in bits per second) and for calculating the UBRR value for each mode of operation using an internally generated clock source.

Table 52. Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps).

BAUD Baud rate (in bits per second, bps)

f_{osc} System Oscillator clock frequency

UBRR Contents of the UBRRH and UBRRL Registers, (0 - 4095)

Some examples of UBRR values for some system clock frequencies are found in Table 60 (see page 156).

Double Speed Operation (U2X)

The transfer rate can be doubled by setting the U2X bit in UCSRA. Setting this bit only has effect for the asynchronous operation. Set this bit to zero when using synchronous operation.

Setting this bit will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication. Note however that the Receiver will in this case only use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery, and therefore a more accurate baud rate setting and system clock are required when this mode is used. For the Transmitter, there are no downsides.

External Clock

External clocking is used by the Synchronous Slave modes of operation. The description in this section refers to Figure 62 for details.

External clock input from the XCK pin is sampled by a synchronization register to minimize the chance of meta-stability. The output from the synchronization register must then pass through an edge detector before it can be used by the Transmitter and Receiver. This process introduces a two CPU clock period delay and therefore the maximum external XCK clock frequency is limited by the following equation:

$$f_{XCK} < \frac{f_{OSC}}{4}$$

Note that f_{osc} depends on the stability of the system clock source. It is therefore recommended to add some margin to avoid possible loss of data due to frequency variations.

Accessing UBRRH/UCSRC Registers

The UBRRH Register shares the same I/O location as the UCSRC Register. Therefore some special consideration must be taken when accessing this I/O location.

Write Access

When doing a write access of this I/O location, the high bit of the value written, the USART Register Select (URSEL) bit, controls which one of the two registers that will be written. If URSEL is zero during a write operation, the UBRRH value will be updated. If URSEL is one, the UCSRC setting will be updated.

The following code examples show how to access the two registers.

Assembly Code Examples[1]

```
...
; Set UBRRH to 2
ldi r16,0x02
out UBRRH,r16
...
; Set the USBS and the UCSZ1 bit to one, and
; the remaining bits to zero.
ldi r16,(1<<URSEL)|(1<<USBS)|(1<<UCSZ1)
out UCSRC,r16
...
```

C Code Examples[1]

```
...
/* Set UBRRH to 2 */
UBRRH = 0x02;
...
/* Set the USBS and the UCSZ1 bit to one, and */
/* the remaining bits to zero. */
UCSRC = (1<<URSEL)|(1<<USBS)|(1<<UCSZ1);
...
```

Note: 1. The example code assumes that the part specific header file is included.

As the code examples illustrate, write accesses of the two registers are relatively unaffected of the sharing of I/O location.

Read Access

Doing a read access to the UBRRH or the UCSRC Register is a more complex operation. However, in most applications, it is rarely necessary to read any of these registers.

The read access is controlled by a timed sequence. Reading the I/O location once returns the UBRRH Register contents. If the register location was read in previous system clock cycle, reading the register in the current clock cycle will return the UCSRC contents. Note that the timed sequence for reading the UCSRC is an atomic operation. Interrupts must therefore be controlled (e.g., by disabling interrupts globally) during the read operation.

The following code example shows how to read the UCSRC Register contents.

Assembly Code Example[1]

```
USART_ReadUCSRC:
  ; Read UCSRC
  in  r16,UBRRH
  in  r16,UCSRC
  ret
```

C Code Example[1]

```
unsigned char USART_ReadUCSRC( void )
{
  unsigned char ucsrc;
  /* Read UCSRC */
  ucsrc = UBRRH;
  ucsrc = UCSRC;
  return ucsrc;
}
```

Note: 1. The example code assumes that the part specific header file is included.

The assembly code example returns the UCSRC value in r16.

Reading the UBRRH contents is not an atomic operation and therefore it can be read as an ordinary register, as long as the previous instruction did not access the register location.

USART Register Description

USART I/O Data Register – UDR

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

The transmit buffer can only be written when the UDRE Flag in the UCSRA Register is set. Data written to UDR when the UDRE Flag is not set, will be ignored by the USART Transmitter. When data is written to the transmit buffer, and the Transmitter is enabled, the Transmitter will load the data into the Transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxD pin.

The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use Read-Modify-Write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS), since these also will change the state of the FIFO.

USART Control and Status Register A – UCSRA

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- **Bit 7 – RXC: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e. does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXC bit will become zero. The RXC Flag can be used to generate a Receive Complete interrupt (see description of the RXCIE bit).

- **Bit 6 – TXC: USART Transmit Complete**

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC Flag can generate a Transmit Complete interrupt (see description of the TXCIE bit).

- **Bit 5 – UDRE: USART Data Register Empty**

The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty interrupt (see description of the UDRIE bit).

UDRE is set after a reset to indicate that the Transmitter is ready.

- **Bit 4 – FE: Frame Error**

This bit is set if the next character in the receive buffer had a Frame Error when received (i.e., when the first stop bit of the next character in the receive buffer is zero). This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

- **Bit 3 – DOR: Data OverRun**

This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 2 – PE: Parity Error**

This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 1 – U2X: Double the USART transmission speed**

This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation.

Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- **Bit 0 – MPCM: Multi-processor Communication Mode**

This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting. For more detailed information see "Multi-processor Communication Mode" on page 148.

USART Control and Status Register B – UCSRB

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – RXCIE: RX Complete Interrupt Enable**

Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set.

- **Bit 6 – TXCIE: TX Complete Interrupt Enable**

Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set.

- **Bit 5 – UDRIE: USART Data Register Empty Interrupt Enable**

Writing this bit to one enables interrupt on the UDRE Flag. A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set.

- **Bit 4 – RXEN: Receiver Enable**

Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FE, DOR and PE Flags.

- **Bit 3 – TXEN: Transmitter Enable**

Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled. The disabling of the Transmitter (writing TXEN to zero) will not become effective until ongoing and pending transmissions are completed (i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted). When disabled, the Transmitter will no longer override the TxD port.

- **Bit 2 – UCSZ2: Character Size**

The UCSZ2 bits combined with the UCSZ1:0 bit in UCSRC sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

- **Bit 1 – RXB8: Receive Data Bit 8**

RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR.

- **Bit 0 – TXB8: Transmit Data Bit 8**

TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR.

USART Control and Status Register C – UCSRC

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

The UCSRC Register shares the same I/O location as the UBRRH Register. See the "Accessing UBRRH/UCSRC Registers" on page 149 section which describes how to access this register.

- **Bit 7 – URSEL: Register Select**

This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.

- **Bit 6 – UMSEL: USART Mode Select**

This bit selects between Asynchronous and Synchronous mode of operation.

Table 55. UMSEL Bit Settings

UMSEL	Mode
0	Asynchronous Operation
1	Synchronous Operation

- **Bit 5:4 – UPM1:0: Parity Mode**

These bits enable and set type of Parity Generation and Check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the PE Flag in UCSRA will be set.

Table 56. UPM Bits Settings

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

- **Bit 3 – USBS: Stop Bit Select**

This bit selects the number of stop bits to be inserted by the trAnsmitter. The Receiver ignores this setting.

Table 57. USBS Bit Settings

USBS	Stop Bit(s)
0	1-bit
1	2-bit

- **Bit 2:1 – UCSZ1:0: Character Size**

The UCSZ1:0 bits combined with the UCSZ2 bit in UCSRB sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

Table 58. UCSZ Bits Settings

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

- **Bit 0 – UCPOL: Clock Polarity**

This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

Table 59. UCPOL Bit Settings

UCPOL	Transmitted Data Changed (Output of TxD Pin)	Received Data Sampled (Input on RxD Pin)
0	Rising XCK Edge	Falling XCK Edge
1	Falling XCK Edge	Rising XCK Edge

USART Baud Rate Registers – UBRRL and UBRRHs

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

The UBRRH Register shares the same I/O location as the UCSRC Register. See the "Accessing UBRRH/UCSRC Registers" on page 149 section which describes how to access this register.

- **Bit 15 – URSEL: Register Select**

This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

- **Bit 14:12 – Reserved Bits**

These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRH is written.

- **Bit 11:0 – UBRR11:0: USART Baud Rate Register**

This is a 12-bit register which contains the USART baud rate. The UBRRH contains the four most significant bits, and the UBRRL contains the eight least significant bits of the USART baud rate. Ongoing transmissions by the Transmitter and Receiver will be corrupted if the baud rate is changed. Writing UBRRL will trigger an immediate update of the baud rate prescaler.

Table 62. Examples of UBRR Settings for Commonly Used Oscillator Frequencies (Continued)

Baud Rate (bps)	f_{osc} = 8.0000 MHz				f_{osc} = 11.0592 MHz				f_{osc} = 14.7456 MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	**207**	**0.2%**	**416**	**-0.1%**	**287**	**0.0%**	**575**	**0.0%**	**383**	**0.0%**	**767**	**0.0%**
4800	**103**	**0.2%**	**207**	**0.2%**	**143**	**0.0%**	**287**	**0.0%**	**191**	**0.0%**	**383**	**0.0%**
9600	**51**	**0.2%**	**103**	**0.2%**	**71**	**0.0%**	**143**	**0.0%**	**95**	**0.0%**	**191**	**0.0%**
14.4k	34	-0.8%	68	0.6%	**47**	**0.0%**	**95**	**0.0%**	**63**	**0.0%**	**127**	**0.0%**
19.2k	**25**	**0.2%**	**51**	**0.2%**	**35**	**0.0%**	**71**	**0.0%**	**47**	**0.0%**	**95**	**0.0%**
28.8k	16	2.1%	34	-0.8%	**23**	**0.0%**	**47**	**0.0%**	**31**	**0.0%**	**63**	**0.0%**
38.4k	**12**	**0.2%**	**25**	**0.2%**	**17**	**0.0%**	**35**	**0.0%**	**23**	**0.0%**	**47**	**0.0%**
57.6k	8	-3.5%	16	2.1%	**11**	**0.0%**	**23**	**0.0%**	**15**	**0.0%**	**31**	**0.0%**
76.8k	6	-7.0%	**12**	**0.2%**	**8**	**0.0%**	**17**	**0.0%**	**11**	**0.0%**	**23**	**0.0%**
115.2k	3	8.5%	8	-3.5%	**5**	**0.0%**	**11**	**0.0%**	**7**	**0.0%**	**15**	**0.0%**
230.4k	1	8.5%	3	8.5%	**2**	**0.0%**	**5**	**0.0%**	**3**	**0.0%**	**7**	**0.0%**
250k	**1**	**0.0%**	**3**	**0.0%**	2	-7.8%	5	-7.8%	3	-7.8%	6	5.3%
0.5M	**0**	**0.0%**	**1**	**0.0%**	–	–	2	-7.8%	1	-7.8%	3	-7.8%
1M	–	–	**0**	**0.0%**	–	–	–	–	0	-7.8%	1	-7.8%
Max (1)	0.5 Mbps		1 Mbps		691.2 kbps		1.3824 Mbps		921.6 kbps		1.8432 Mbps	

1. UBRR = 0, Error = 0.0%

Electrical Characteristics

Note: Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Min and Max values will be available after the device is characterized.

Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin except $\overline{\text{RESET}}$ with respect to Ground	-0.5V to V_{CC}+0.5V
Voltage on $\overline{\text{RESET}}$ with respect to Ground	-0.5V to +13.0V
Maximum Operating Voltage	6.0V
DC Current per I/O Pin	40.0 mA
DC Current V_{CC} and GND Pins	200.0 mA

*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

DC Characteristics

T_A = -40°C to 85°C, V_{CC} = 2.7V to 5.5V (unless otherwise noted)

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{IL}	Input Low Voltage	Except XTAL1 pin	-0.5		0.2 V_{CC}[1]	V
V_{IL1}	Input Low Voltage	XTAL1 pin, External Clock Selected	-0.5		0.1 V_{CC}[1]	V
V_{IH}	Input High Voltage	Except XTAL1 and $\overline{\text{RESET}}$ pins	0.6 V_{CC}[2]		V_{CC} + 0.5	V
V_{IH1}	Input High Voltage	XTAL1 pin, External Clock Selected	0.8 V_{CC}[2]		V_{CC} + 0.5	V
V_{IH2}	Input High Voltage	$\overline{\text{RESET}}$ pin	0.9 V_{CC}[2]		V_{CC} + 0.5	V
V_{OL}	Output Low Voltage[3] (Ports A,B,C,D)	I_{OL} = 20 mA, V_{CC} = 5V I_{OL} = 10 mA, V_{CC} = 3V			0.7 0.5	V V
V_{OH}	Output High Voltage[4] (Ports A,B,C,D)	I_{OH} = -20 mA, V_{CC} = 5V I_{OH} = -10 mA, V_{CC} = 3V	4.2 2.2			V V
I_{IL}	Input Leakage Current I/O Pin	Vcc = 5.5V, pin low (absolute value)			1	µA
I_{IH}	Input Leakage Current I/O Pin	Vcc = 5.5V, pin high (absolute value)			1	µA
R_{RST}	Reset Pull-up Resistor		30		80	kΩ
R_{pu}	I/O Pin Pull-up Resistor		20		50	kΩ

T_A = -40°C to 85°C, V_{CC} = 2.7V to 5.5V (unless otherwise noted) (Continued)

Symbol	Parameter	Condition	Min	Typ	Max	Units
I_{CC}	Power Supply Current	Active 4 MHz, V_{CC} = 3V (ATmega8L)			5	mA
		Active 8 MHz, V_{CC} = 5V (ATmega8)			15	mA
		Idle 4 MHz, V_{CC} = 3V (ATmega8L)			2	mA
		Idle 8 MHz, V_{CC} = 5V (ATmega8)			7	mA
	Power-down mode(5)	WDT enabled, V_{CC} = 3V			25	µA
		WDT disabled, V_{CC} = 3V			2	µA
V_{ACIO}	Analog Comparator Input Offset Voltage	V_{CC} = 5V V_{in} = V_{CC}/2			20	mV
I_{ACLK}	Analog Comparator Input Leakage Current	V_{CC} = 5V V_{in} = V_{CC}/2	-50		50	nA
t_{ACID}	Analog Comparator Propagation Delay	V_{CC} = 2.7V V_{CC} = 4.0V		750 500		ns

Notes:
1. "Max" means the highest value where the pin is guaranteed to be read as low
2. "Min" means the lowest value where the pin is guaranteed to be read as high
3. Although each I/O port can sink more than the test conditions (20mA at Vcc = 5V, 10mA at Vcc = 3V) under steady state conditions (non-transient), the following must be observed:
   PDIP Package:
   1] The sum of all IOL, for all ports, should not exceed 400 mA.
   2] The sum of all IOL, for ports C0 - C5 should not exceed 200 mA.
   3] The sum of all IOL, for ports B0 - B7, C6, D0 - D7 and XTAL2, should not exceed 100 mA.
   TQFP and MLF Package:
   1] The sum of all IOL, for all ports, should not exceed 400 mA.
   2] The sum of all IOL, for ports C0 - C5, should not exceed 200 mA.
   3] The sum of all IOL, for ports C6, D0 - D4, should not exceed 300 mA.
   4] The sum of all IOL, for ports B0 - B7, D5 - D7, should not exceed 300 mA.
   If IOL exceeds the test condition, VOL may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test condition.
4. Although each I/O port can source more than the test conditions (20mA at Vcc = 5V, 10mA at Vcc = 3V) under steady state conditions (non-transient), the following must be observed:
   PDIP Package:
   1] The sum of all IOH, for all ports, should not exceed 400 mA.
   2] The sum of all IOH, for port C0 - C5, should not exceed 100 mA.
   3] The sum of all IOH, for ports B0 - B7, C6, D0 - D7 and XTAL2, should not exceed 100 mA.
   TQFP and MLF Package:
   1] The sum of all IOH, for all ports, should not exceed 400 mA.
   2] The sum of all IOH, for ports C0 - C5, should not exceed 200 mA.
   3] The sum of all IOH, for ports C6, D0 - D4, should not exceed 300 mA.
   4] The sum of all IOH, for ports B0 - B7, D5 - D7, should not exceed 300 mA.
   If IOH exceeds the test condition, VOH may exceed the related specification. Pins are not guaranteed to source current greater than the listed test condition.
5. Minimum V_{CC} for Power-down is 2.5V.

Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	9
0x3E (0x5E)	SPH	–	–	–	–	–	SP10	SP9	SP8	11
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
0x3C (0x5C)	Reserved									
0x3B (0x5B)	GICR	INT1	INT0	–	–	–	–	IVSEL	IVCE	47, 65
0x3A (0x5A)	GIFR	INTF1	INTF0	–	–	–	–	–	–	66
0x39 (0x59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	70, 100, 120
0x38 (0x58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0	71, 101, 120
0x37 (0x57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	210
0x36 (0x56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	168
0x35 (0x55)	MCUCR	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	31, 64
0x34 (0x54)	MCUCSR	–	–	–	–	WDRF	BORF	EXTRF	PORF	39
0x33 (0x53)	TCCR0	–	–	–	–	–	CS02	CS01	CS00	70
0x32 (0x52)	TCNT0	Timer/Counter0 (8 Bits)								70
0x31 (0x51)	OSCCAL	Oscillator Calibration Register								29
0x30 (0x50)	SFIOR	–	–	–	–	ACME	PUD	PSR2	PSR10	56, 73, 121, 190
0x2F (0x4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	95
0x2E (0x4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	98
0x2D (0x4D)	TCNT1H	Timer/Counter1 – Counter Register High byte								99
0x2C (0x4C)	TCNT1L	Timer/Counter1 – Counter Register Low byte								99
0x2B (0x4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High byte								99
0x2A (0x4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low byte								99
0x29 (0x49)	OCR1BH	Timer/Counter1 – Output Compare Register B High byte								99
0x28 (0x48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low byte								99
0x27 (0x47)	ICR1H	Timer/Counter1 – Input Capture Register High byte								100
0x26 (0x46)	ICR1L	Timer/Counter1 – Input Capture Register Low byte								100
0x25 (0x45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	115
0x24 (0x44)	TCNT2	Timer/Counter2 (8 Bits)								117
0x23 (0x43)	OCR2	Timer/Counter2 Output Compare Register								117
0x22 (0x42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB	117
0x21 (0x41)	WDTCR	–	–	–	WDCE	WDE	WDP2	WDP1	WDP0	41
0x20[1] (0x40)[1]	UBRRH	URSEL	–	–	–	UBRR[11:8]				155
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	153
0x1F (0x3F)	EEARH	–	–	–	–	–	–	–	EEAR8	18
0x1E (0x3E)	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	18
0x1D (0x3D)	EEDR	EEPROM Data Register								18
0x1C (0x3C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE	18
0x1B (0x3B)	Reserved									
0x1A (0x3A)	Reserved									
0x19 (0x39)	Reserved									
0x18 (0x38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	63
0x17 (0x37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	63
0x16 (0x36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	63
0x15 (0x35)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	63
0x14 (0x34)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	63
0x13 (0x33)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	63
0x12 (0x32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	63
0x11 (0x31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	63
0x10 (0x30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	63
0x0F (0x2F)	SPDR	SPI Data Register								128
0x0E (0x2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X	128
0x0D (0x2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	126
0x0C (0x2C)	UDR	USART I/O Data Register								150
0x0B (0x2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	151
0x0A (0x2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	152
0x09 (0x29)	UBRRL	USART Baud Rate Register Low byte								155
0x08 (0x28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	191
0x07 (0x27)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	202
0x06 (0x26)	ADCSRA	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	204
0x05 (0x25)	ADCH	ADC Data Register High byte								205
0x04 (0x24)	ADCL	ADC Data Register Low byte								205
0x03 (0x23)	TWDR	Two-wire Serial Interface Data Register								170
0x02 (0x22)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	170

Register Summary (Continued)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x01 (0x21)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	170
0x00 (0x20)	TWBR	Two-wire Serial Interface Bit Rate Register								168

Notes:
1. Refer to the USART description for details on how to access UBRRH and UCSRC.
2. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that the CBI and SBI instructions will operate on all bits in the I/O Register, writing a one back into any flag read as set, thus clearing the flag. The CBI and SBI instructions work with registers 0x00 to 0x1F only.

Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← 0xFF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← 0xFF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd – Rr	Z, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H	1
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1 / 2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2
Mnemonics	**Operands**	**Description**	**Operation**	**Flags**	**#Clocks**

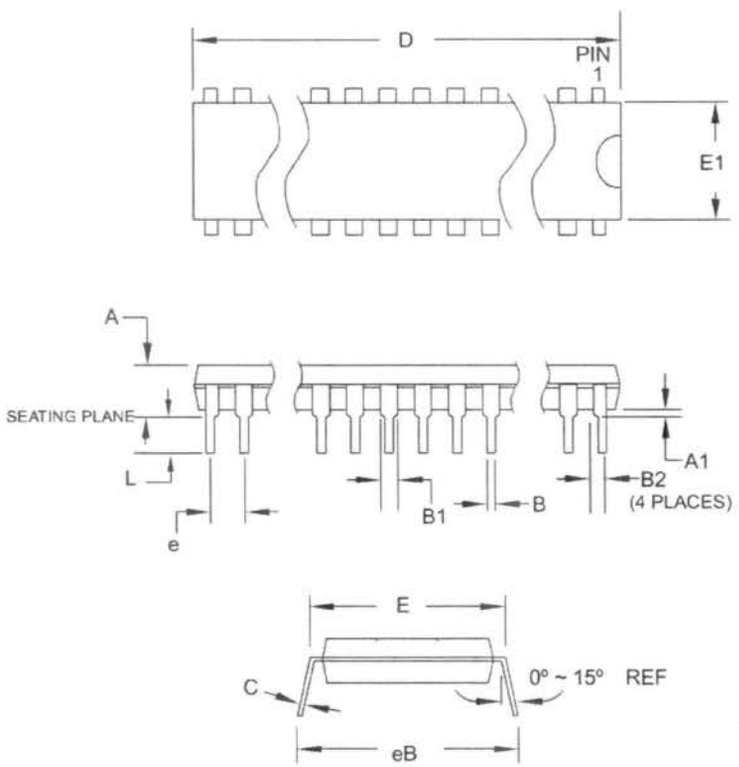
Instruction Set Summary (Continued)

BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store Program Memory	(Z) ← R1:R0	None	-
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2
BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
Mnemonics	**Operands**	**Description**	**Operation**	**Flags**	**#Clocks**

Instruction Set Summary (Continued)

CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
MCU CONTROL INSTRUCTIONS					
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1

28P3



COMMON DIMENSIONS
(Unit of Measure = mm)

SYMBOL	MIN	NOM	MAX	NOTE
A	–	–	4.5724	
A1	0.508	–	–	
D	34.544	–	34.798	Note 1
E	7.620	–	8.255	
E1	7.112	–	7.493	Note 1
B	0.381	–	0.533	
B1	1.143	–	1.397	
B2	0.762	–	1.143	
L	3.175	–	3.429	
C	0.203	–	0.356	
eB	–	–	10.160	
e	2.540 TYP			

Note: 1. Dimensions D and E1 do not include mold Flash or Protrusion. Mold Flash or Protrusion shall not exceed 0.25 mm (0.010").

09/28/01

	TITLE	DRAWING NO.	REV.
2325 Orchard Parkway San Jose, CA 95131	**28P3**, 28-lead (0.300"/7.62 mm Wide) Plastic Dual Inline Package (PDIP)	28P3	B

ΠΑΡΑΡΤΗΜΑ Ά Β

Στο παράρτημα αυτό παραθέτω τα πιο σημαντικά τμήματα από τον οδηγό χρήσης της πλατφόρμας ανάπτυξης πρωτότυπου της Atmel STK 500.

Section 1
Introduction

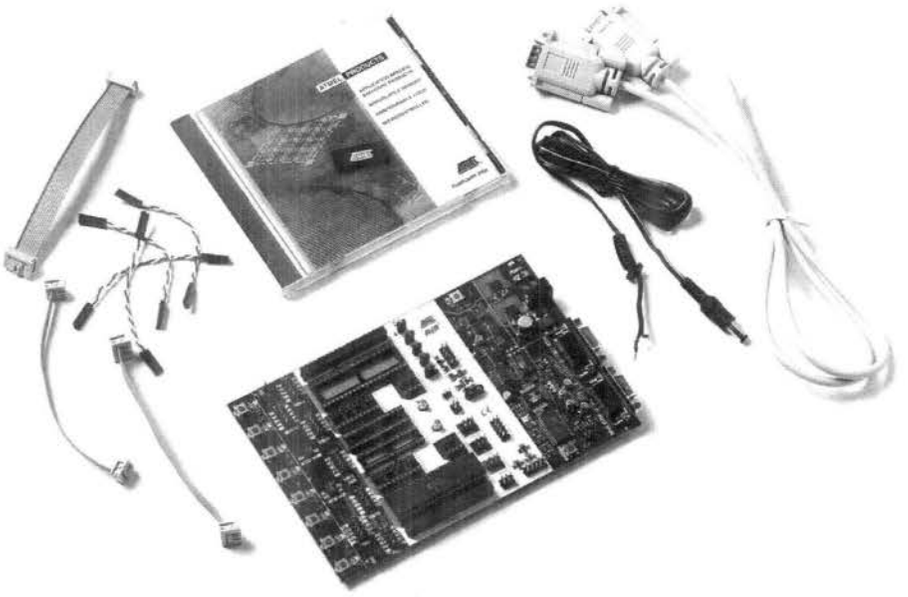
Congratulations on your purchase of the AVR® STK500 Flash Microcontroller Starter Kit. The STK500 is a complete starter kit and development system for the AVR Flash Microcontroller from Atmel Corporation. It is designed to give designers a quick start to develop code on the AVR and for prototyping and testing of new designs.

1.1 Starter Kit Features

- AVR Studio® Compatible
- RS-232 Interface to PC for Programming and Control
- Regulated Power Supply for 10 - 15V DC Power
- Sockets for 8-pin, 20-pin, 28-pin, and 40-pin AVR Devices
- Parallel and Serial High-voltage Programming of AVR Devices
- Serial In-System Programming (ISP) of AVR Devices
- In-System Programmer for Programming AVR Devices in External Target System
- Reprogramming of AVR Devices
- 8 Push Buttons for General Use
- 8 LEDs for General Use
- All AVR I/O Ports Easily Accessible through Pin Header Connectors
- Additional RS-232 Port for General Use
- Expansion Connectors for Plug-in Modules and Prototyping Area
- On-board 2-Mbit DataFlash® for Nonvolatile Data Storage

The STK500 is supported by AVR Studio, version 3.2 or higher. For up-to-date information on this and other AVR tool products, please read the document "avrtools.pdf". The newest version of AVR Studio, "avrtools.pdf" and this user guide can be found in the AVR section of the Atmel web site, *www.atmel.com*.

Figure 1-1. STK500



1.2 Device Support

The system software currently supports the following devices in all speed grades:

- ATtiny11
- ATtiny12
- ATtiny15
- ATtiny22
- ATtiny28
- AT90S1200
- AT90S2313
- AT90S2323
- AT90S2333
- AT90S2343
- AT90S4414
- AT90S4433
- AT90S4434
- AT90S8515
- AT90S8535
- ATmega8
- ATmega16
- ATmega161
- ATmega163
- ATmega323
- ATmega103[1]
- ATmega128[1]

Note: *1.* In external target or in STK501, devices do not fit into the sockets of STK500.

Support for new AVR devices may be added in new versions of AVR Studio. The latest version of AVR Studio is always available from *www.atmel.com*.

Section 2

Getting Started

2.1 Unpacking the System

Kit contents:

- STK500 starter kit evaluation board
- Cables for STK500:
  - (2 pcs) 10-wire cables for I/O ports and parallel mode programming
  - (1 pc) 6-wire cable for In-System Programming
  - (4 pcs) 2-wire cable for UART and DataFlash connections
- 9-pin RS-232 cable
- DC power cable
- Atmel CD-ROM with datasheets and software
- AT90S8515-8PC sample microcontroller

2.2 System Requirements

The minimum hardware and software requirements are:

- 486 processor (Pentium® is recommended)
- 16 MB RAM
- 12 MB free hard disk space (AVR Studio)
- Windows® 95/98/2000/ME and Windows NT® 4.0 or higher
- 115200 baud RS-232 port (COM port)
- 10 - 15V DC power supply, 500 mA min.

2.3 Quick Start

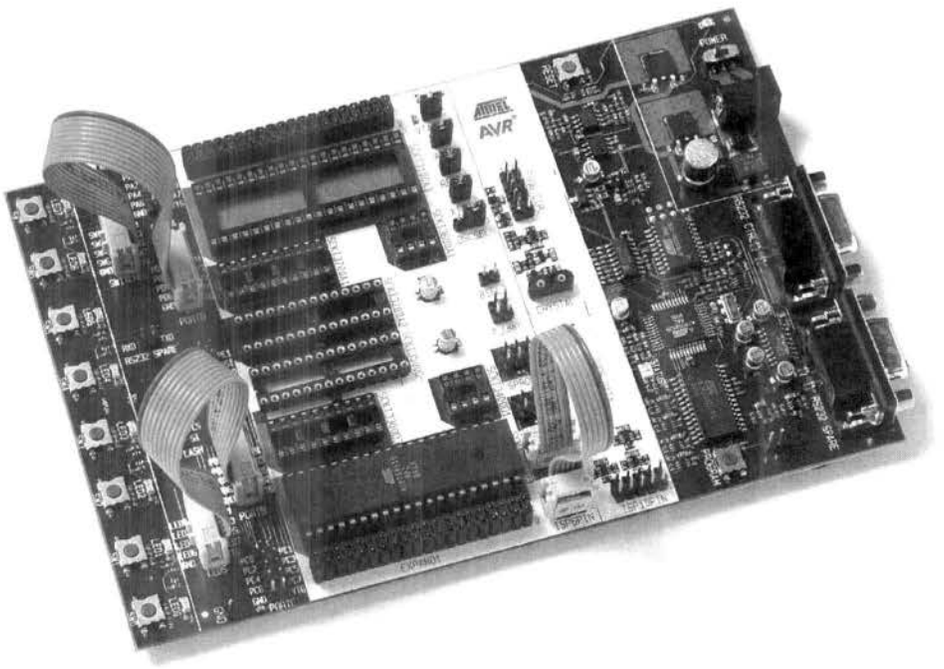
The STK500 starter kit is shipped with an AT90S8515-8PC microcontroller in the socket marked SCKT3000D3. The default jumper settings will allow the microcontroller to execute from the clock source and voltage regulator on the STK500 board.

The microcontroller is programmed with a test program that toggles the LEDs. The test program in the AT90S8515 is similar to the example application code described in Section 9. Connect the LEDs and switches and power up the STK500 to run the test program in the AT90S8515.

Use the supplied 10-pin cables to connect the header marked "PORTB" with the header marked "LEDS", and connect the header marked "PORTD" with the header marked "SWITCHES". The connections are shown in Figure 2-1.

An external 10 - 15V DC power supply is required. The input circuit is a full bridge rectifier, and the STK500 automatically handles both positive or negative center connectors. If a positive center connector is used, it can be impossible to turn the STK500 off since the power switch disconnects the GND terminal. In this case, GND can be supplied through the RS-232 cable shield if connected or through alternative GND connections. Connect the power cable between a power supply and the STK500. Apply 10 - 15V DC to the power connector. The power switch turns the STK500 main power on and off. The red LED is lit when power is on, and the status LEDs will go from red, via yellow, to green. The green LED indicates that the target V_{CC} is present. The program now running in the AT90S8515 will respond to pressed switches by toggling the LEDs.

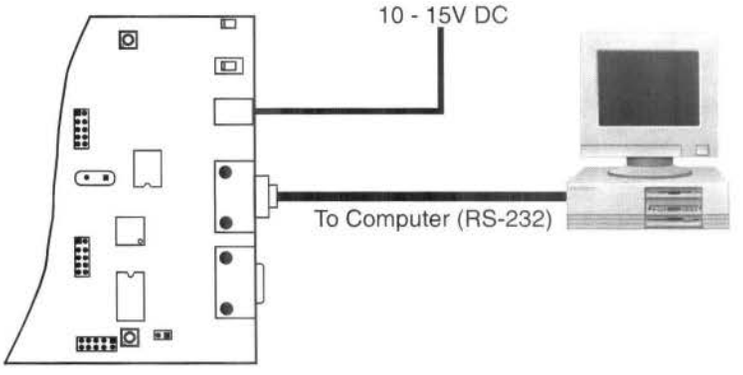
Figure 2-1. Default Setup of STK500



The starter kit can be configured for various clock and power sources. A complete description of the jumper settings is explained in paragraph 3.8 on page 3-15 and on the reverse side of the starter kit.

2.3.1 Connecting the Hardware

Figure 2-2. Connection to STK500



To program the AT90S8515, connect the supplied 6-wire cable between the ISP6PIN header and the SPROG3 target ISP header as shown in Figure 2-1. Section 3.7.1 on page 3-9 describes the programming cable connections.

Connect a serial cable to the connector marked "RS232 CTRL" on the evaluation board to a COM port on the PC as shown in Figure 2-2. Install AVR Studio software on the PC. Instructions on how to install and use AVR Studio are given in Section 5 on page 5-1. When AVR Studio is started, the program will automatically detect to which COM port the STK500 is connected.

2.3.2 Programming the Target AVR Device

The STK500 is controlled from AVR Studio, version 3.2 and higher. AVR Studio is an integrated development environment (IDE) for developing and debugging AVR applications. AVR Studio provides a project management tool, source file editor, simulator, in-circuit emulator interface and programming interface for STK500.

To program a hex file into the target AVR device, select "STK500" from the "Tools" menu in AVR Studio.

Select the AVR target device from the pull-down menu on the "Program" tab and locate the intel-hex file to download.

Press the "Erase" button, followed by the "Program" button. The status LED will now turn yellow while the part is programmed, and when programming succeeds, the LED will turn green. If programming fails, the LED will turn red after programming. See the troubleshooting guide in Section 7 on page 7-1.

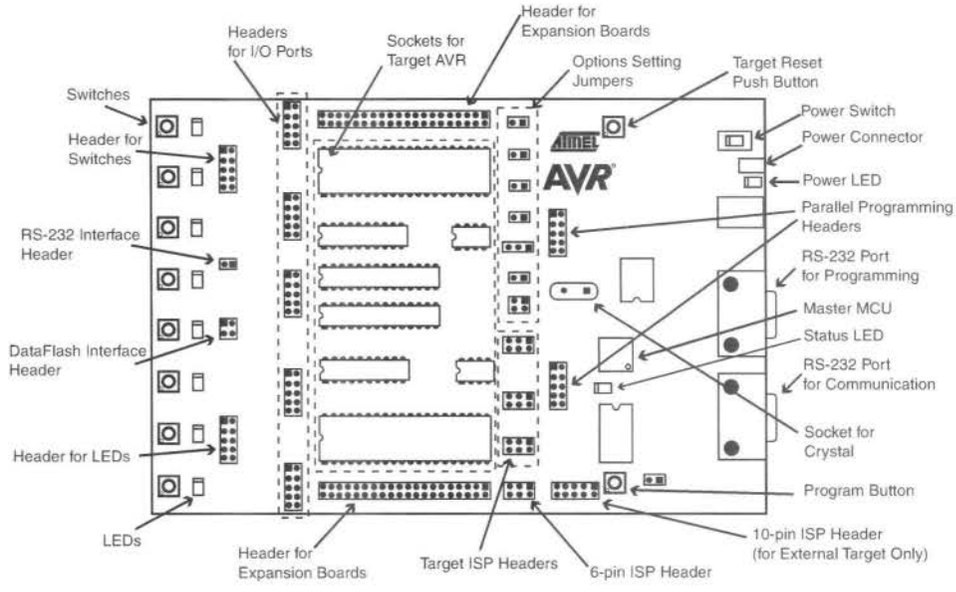
Figure 2-3. AVR Studio STK500 Programming Menu

Complete descriptions of using the STK500 interface in AVR Studio are given in Section 5 on page 5-1.

Section 3

Hardware Description

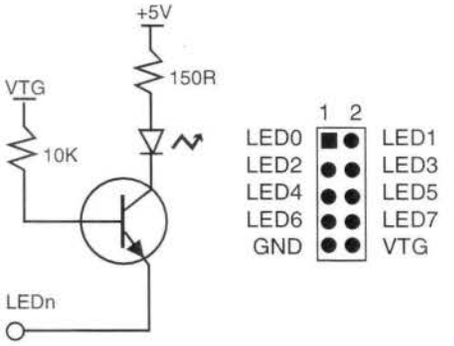
Figure 3-1. STK500 Components



3.1 Description of User LEDs

The STK500 starter kit includes 8 yellow LEDs and 8 push-button switches. The LEDs and switches are connected to debug headers that are separated from the rest of the board. They can be connected to the AVR devices with the supplied 10-wire cable to the pin header of the AVR I/O ports. Figure 3-4 shows how the LEDs and switches can be connected to the I/O port headers. The cables should be connected directly from the port header to the LED or switch header. The cable should not be twisted. A red wire on the cable indicates pin 1. Confirm that this is connected to pin 1 on each of the headers. Figure 3-2 shows how the LED control is implemented. This solution will give the same amount of light from the LED for all target voltages from 1.8V to 6.0V.

Figure 3-2. Implementation of LEDs and LED Headers



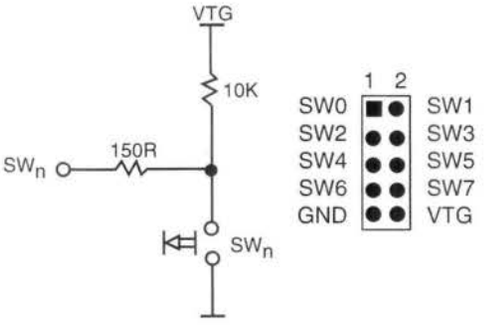
Note: The AVR can source or sink enough current to drive a LED directly. In the STK500 design, a transistor with two resistors is used to give the same amount of light from the LED, whatever the target voltage (VTG) may be and to turn off the LEDs when VTG is missing.

3.2 Description of User Switches

The switches connected to the debug headers are implemented as shown in Figure 3-3.

Pushing a switch causes the corresponding SWx to be pulled low, while releasing it will result in VTG on the appropriate switch header connector. Valid target voltage range is 1.8V < VTG < 6.0V.

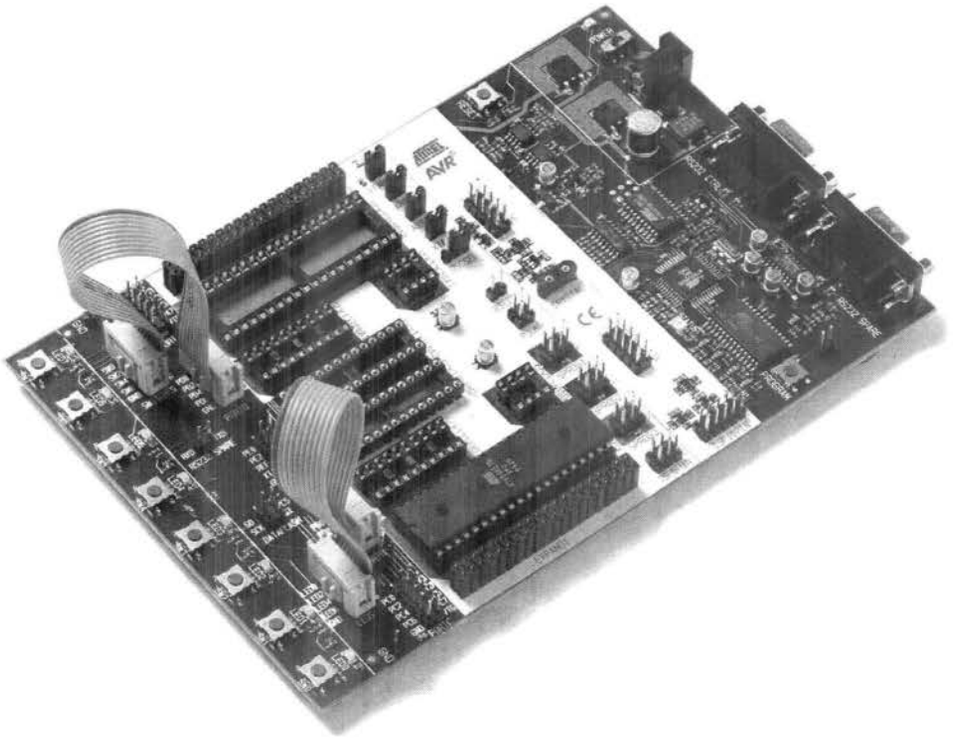
Figure 3-3. Implementation of Switches and Switch Headers



Note: In the AVR, the user can enable internal pull-ups on the input pins, removing the need for an external pull-up on the push-button. In the STK500 design, we have added an external 10K pull-up to give all users a logical "1" on SWn when the push-button is not pressed. The 150R resistor limits the current going into the AVR.

3.3 Connection of LEDs and Switches

Figure 3-4. Connection of LEDs and Switches to I/O Port Headers

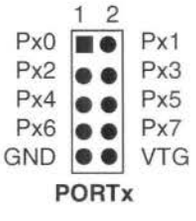


Any I/O port of the AVR can be connected to the LEDs and switches using the 10-wire cables. The headers are supplied with VTG (target V_{CC}) and GND lines in addition to the signal lines.

3.4 Port Connectors

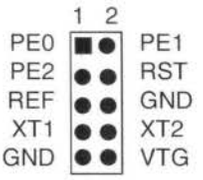
The pinout for the I/O port headers is explained in Figure 3-5. The square marking indicates pin 1.

Figure 3-5. General Pinout of I/O Port Headers



The PORTE/AUX header has some special signals and functions in addition to the PORTE pins. The pinout of this header is shown in Figure 3-6.

Figure 3-6. Pinout of PORTE Header



The special functions of this port are:

- **PE0 - PE2:**

Table 3-1. PORTE Connection

	ATmega161	AT90S4414/AT90S8515
PE0	PE0/ICP/INT2	ICP
PE1	PE1/ALE	ALE
PE2	PE2/OC1B	OC1B

- **REF:** Analog reference voltage. This pin is connected to the AREF pin on devices having a separate analog reference pin.
- **XT1:** XTAL 1 pin. The internal main clock signal to all sockets. If the XTAL1 jumper is disconnected, this pin can be used as external clock signal.
- **XT2:** XTAL 2 pin. If the XTAL1 jumper is disconnected, this pin can be used for external crystal with the XT1 pin.

The headers for the LEDs and switches use the same pinout as the I/O port headers. The pinout of the switch header is explained in Figure 3-7 and the pinout for the LED header is explained in Figure 3-8. The square marking indicates pin 1.

Figure 3-7. Pinout of the Switch Header

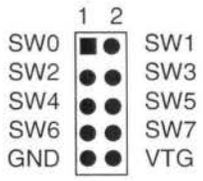
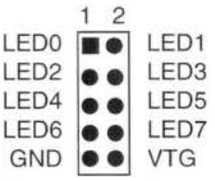


Figure 3-8. Pinout of the LED Header



3.5 Description of User RS-232 Interface

The STK500 includes two RS-232 ports. One RS-232 port is used for communicating with AVR Studio. The other RS-232 can be used for communicating between the target AVR microcontroller in the socket and a PC serial port connected to the RS-232. To use the RS-232, the UART pins of the AVR need to be physically connected to the RS-232.

The 2-pin header marked "RS232 SPARE" can be used for connecting the RS-232 converter to the UART pins on the target AVR microcontroller in the socket. Use the 2-wire cable to connect the UART pins to the RS-232. The connection is shown in Figure 3-9. The block schematic of the RS-232 connection is shown in Figure 3-10.

Figure 3-9. Connection of I/O Pins to UART

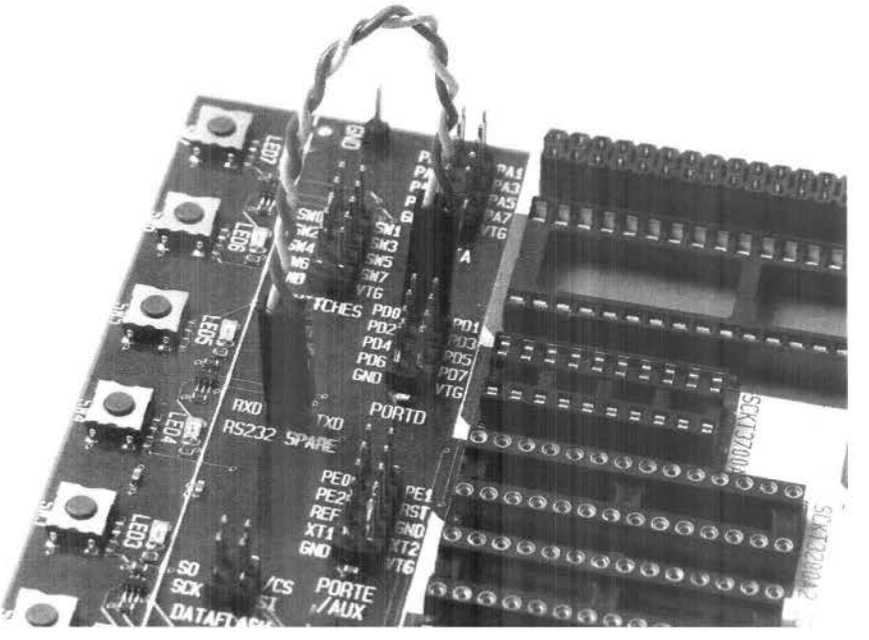
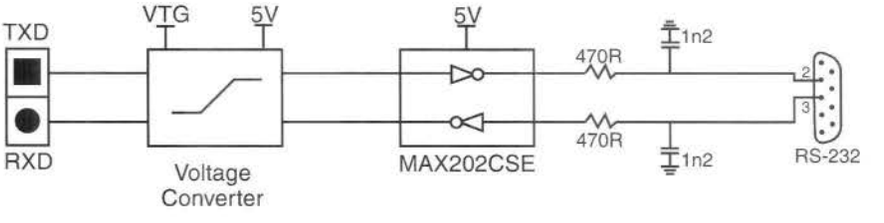


Figure 3-10. Schematic of UART Pin Connections



3.7 Target Socket Section

The programming module consists of the eight sockets in the white area in the middle of the starter kit. In these sockets, the target AVR devices can be inserted for programming and are used in the application.

Note: Only one AVR device should be inserted in the sockets at a time.

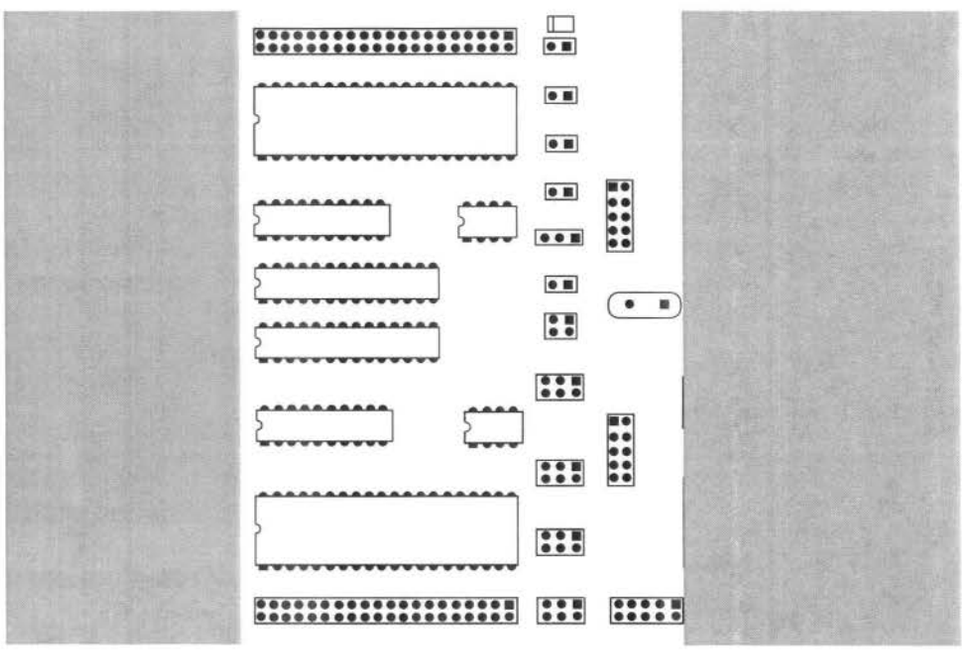
The AVR Flash memory is guaranteed to be correct after 1,000 programming operations; the typical lifetime of the Flash memory is much longer.

Note: When inserting a device in the socket, notice the orientation of the device.

The notch on the short side of the part must match the notch on the socket. If the device is inserted the wrong way, it may damage the part and the starter kit.

The socket section is used for both running applications and target device programming.

Figure 3-15. The STK500 Programming Module



The part inserted in the socket can be programmed in the system from AVR Studio with two different methods:

1. AVR In-System Programming (ISP) running at the parts normal supply voltage.
2. High-voltage Programming, where the supply voltage is always 5 volts.

Four general nets (VTARGET, RESET, XTAL1 and AREF) can be connected to the socket section.

The following sections describe how to use both programming methods. For instructions on using the AVR Studio programming software, see Section 5, "Using AVR Studio" on page 5-1.

3.7.1 ISP Programming

In-System Programming uses the AVR internal SPI (Serial Peripheral Interface) to download code into the Flash and EEPROM memory of the AVR. ISP programming requires only V_{CC}, GND, RESET and three signal lines for programming. All AVR devices except AT90C8534, ATtiny11 and ATtiny28 can be ISP programmed. The AVR can be programmed at the normal operating voltage, normally 2.7 - 6.0V. No high-voltage signals are required. The ISP programmer can program both the internal Flash and EEPROM. It also programs fuse bits for selecting clock options, start-up time and internal Brown-out Detector (BOD) for most devices.

High-voltage programming can also program devices that are not supported by ISP programming. Some devices require High-voltage Programming for programming certain fuse bits. See the High-voltage Programming section on page 3-11 for instructions on how to use High-voltage Programming.

Because the programming interface is placed on different pins from part to part, three programming headers are used to route the programming signals to the correct pins. A 6-wire cable is supplied for connecting the ISP signals to the target ISP header. A color coding system and a number system are used to explain which target ISP header is used for each socket.

During ISP programming, the 6-wire cable must always be connected to the header marked "ISP6PIN". When programming parts in the blue sockets, connect the other end of the cable to the blue SPROG1 target ISP header. When programming parts in the green socket, use the green SPROG2 target ISP header. And when programming parts in the red sockets, use the red SPROG3 target ISP header.Table 3-2 shows which socket suits which AVR device, and which SPROG target ISP header to use for ISP programming.

The 6-wire cables should be connected directly from the ISP6PIN header to the correct SPROG target ISP header. The cable should not be twisted. A colored wire on the cable indicates pin 1. Confirm that this is connected to pin 1 on each of the headers.

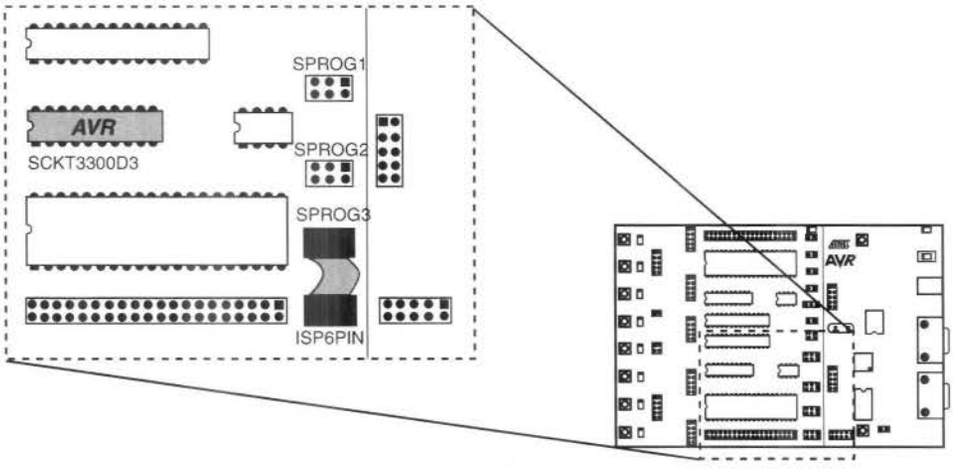
When programming 8-pin devices, note the following: Pin 1 is used both as RESET and as PB5 on some devices (ATtiny11, ATtiny12 and ATtiny15). Pin 1 on the 8-pin sockets SCKT3400D1 and SCKT3400D1 are connected to PB5. The RESET signal used during ISP programming is therefore not connected to pin 1 on these sockets. This signal must be connected by placing a wire between RST and the PORTE header and PB5 on the PORTB header.

Table 3-2. AVR Sockets

AVR Devices	STK500 Socket	Color	Number	Target ISP Header
AT90S1200 AT90S2313	SCKT3300D3	Red	3	SPROG3
AT90S2323 AT90S2343 ATtiny12 ATtiny22	SCKT3400D1	Blue	1	SPROG1. Connect RST on PORTE to PB5 on PORTB. Connect XTI on PORTE to PB3 (XTAL1 on 2323) on PORTB.
ATtiny11	SCKT3400D1	Blue	1	High-voltage Programming only
ATtiny28	SCKT3500D-	None	–	High-voltage Programming only
AT90S4414 AT90S8515 ATmega161	SCKT3000D3	Red	3	SPROG3
AT90S4434 AT90S8535 ATmega16 ATmega163 ATmega323	SCKT3100A3	Red	3	SPROG3
AT90S2333 AT90S4433 ATmega8	SCKT3200A2	Green	2	SPROG2
ATtiny15	SCKT3600A1	Blue	1	SPROG1. Connect RST on PORTE to PB5 on PORTB.
N/A	SCKT3700A1	Blue	1	Socket is not in use in this version of STK500
ATmega103 ATmega128	Use the STK501 Top Module			

Figure 3-16 shows an example of how AT90S2313 can be In-System Programmed. The 6-wire cable is connected from the ISP6PIN header to the red SPROG3 target ISP header, and the AT90S2313 part is inserted in the red socket marked "SCKT3100D3".

Figure 3-16. Example Connection for Programming AT90S2313



Section 5

Using AVR Studio

5.1 Windows Software

In this section, the supporting software for STK500 will be presented and an in-depth description of the available programming options is given.

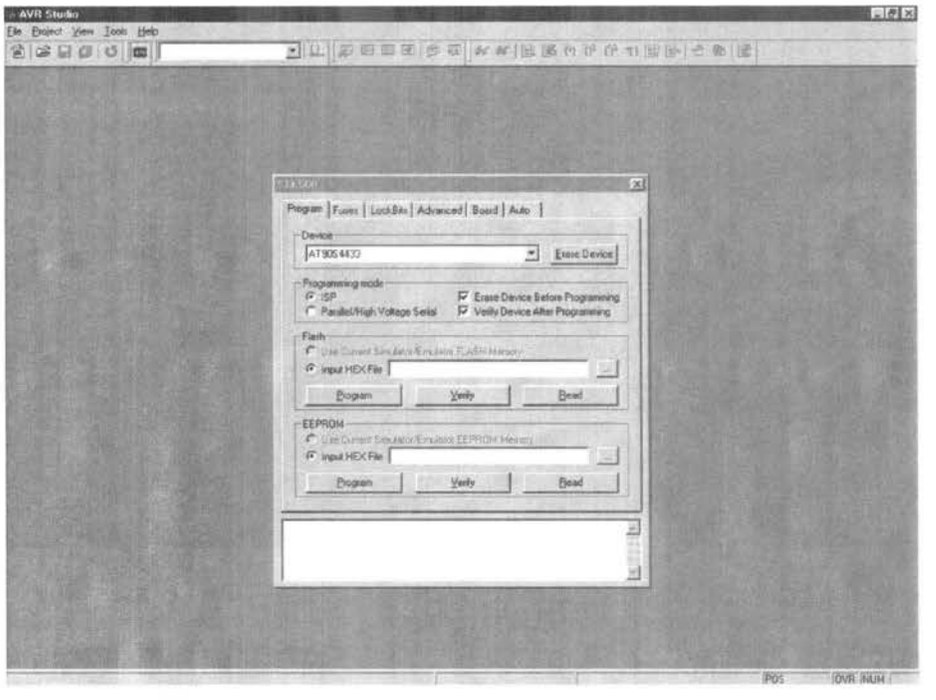
5.2 Starting the Windows Software

The software used for communicating with the STK500 development board is included in AVR Studio, version 3.2 and higher. For information on how to install this software, please see Section 4 on page 4-1. Once installed, AVR Studio can be started by double-clicking on the AVR Studio icon. If default install options are used, the program is located in the Windows "Start menu → Programs → Atmel AVR Tools" folder.

5.2.1 Starting STK500

Pressing the "AVR" button on the AVR Studio toolbar will start the STK500 user interface as shown in Figure 5-1.

Figure 5-1. AVR Studio with STK500 User Interface



5.3 STK500 User Interface

The STK500 user interface includes powerful features for the STK500 development board. The available settings are divided into six groups, each selectable by clicking on the appropriate tab. Since different devices have different features, the available options and selections will depend on which device is selected. Unavailable features are grayed out.

5.3.1 "Program" Settings

The program settings are divided into four different subgroups.

5.3.1.1 Device

A device is selected by selecting the correct device from the pull-down menu. This group also includes a button that performs a chip erase on the selected device, erasing both the Flash and EEPROM memories.

5.3.1.2 Programming Mode

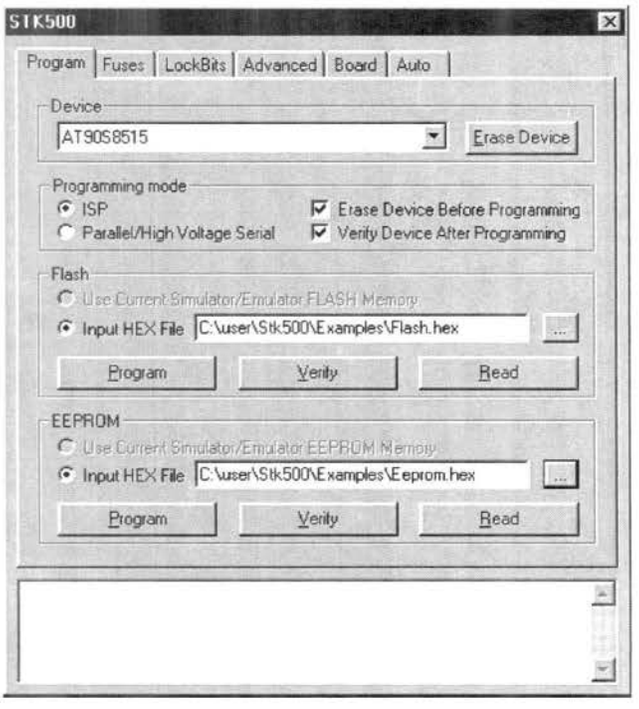
This group selects programming mode. For devices only supporting High-voltage Programming, the ISP option will be grayed out. If both modes are available, select a mode by clicking on the correct method. Checking "Erase Device Before Programming" will force STK500 to perform a chip erase before programming code to the program memory (Flash). Checking "Verify Device After Programming" will force STK500 to perform a verification of the memory after programming it (both Flash and EEPROM).

5.3.1.3 Flash

If the STK500 user interface is opened without a project loaded in AVR Studio, the "Use Current Simulator/Emulator FLASH Memory" option will be grayed out. When a project is open, this option allows programming of the Flash memory content currently present in the Flash Memory view of AVR Studio. For more information about AVR Studio memory views, please take a look in the AVR Studio Help file.

If no project is running, or the source code is stored in a separate hex file, select the "Input HEX File" option. Browse to the correct file by pressing the ... button or type the complete path and filename in the text field. The selected file must be in "Intel-hex" format or "extended Intel-hex" format.

Figure 5-2. Program



5.3.1.4 EEPROM

If the STK500 user interface is opened without a project loaded in AVR Studio, the "Use Current Simulator/Emulator EEPROM Memory" option will be grayed out. When a project is open, this option allows programming of the EEPROM memory content currently present in the EEPROM Memory view. For more information about AVR Studio memory views, please take a look in the AVR Studio Help file.

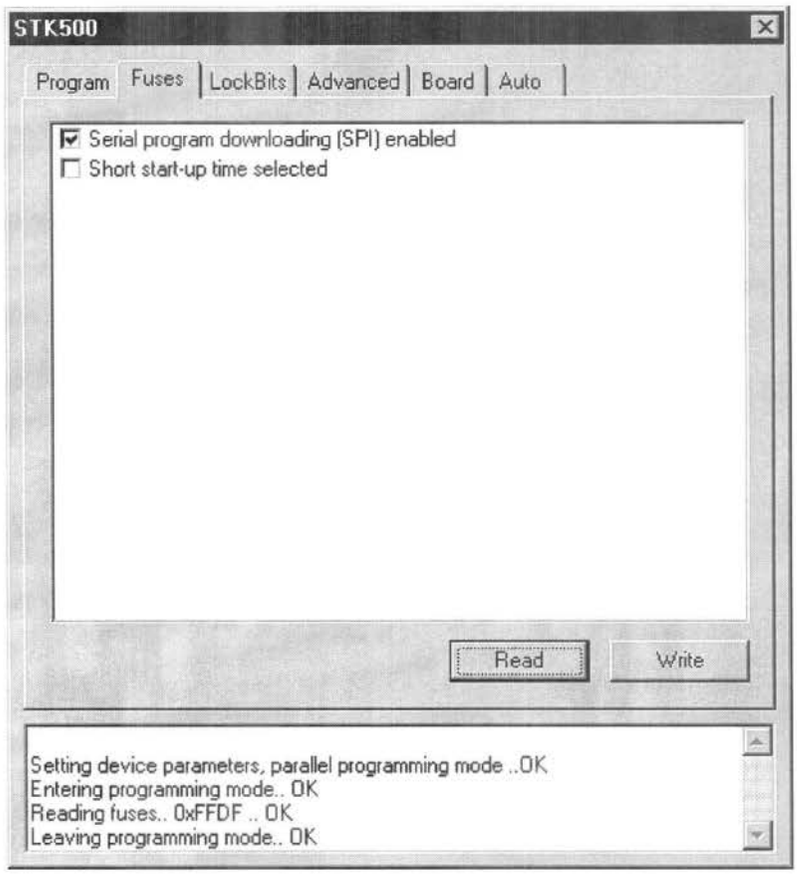
If no project is running, or the source code is stored in a separate hex file, select the "Input HEX File" option. Browse to the correct file by pressing the ... button or type the complete path and filename in the text field. The selected file must be in "Intel-hex" format or "extended Intel-hex" format.

5.3.2 "Fuses" Settings

In the "Fuses" tab an overview of accessible fuses are presented. Some fuses are only available during High-voltage Programming. These will be displayed but not accessible if operating in ISP programming mode. Press the "Read" button to read the current value of the fuses, and the "Write" button to write the current fuse setting to the device. Checking one of these check boxes indicates that this fuse should be enabled/programmed, which means writing a "0" to the fuse location in the actual device. Note that the selected fuse setting is not affected by erasing the device with a chip-erase cycle (i.e., pressing "Chip Erase" button in the "Program" settings).

Detailed information on what fuses are available in the different programming modes and their functions can be found in the appropriate device datasheet.

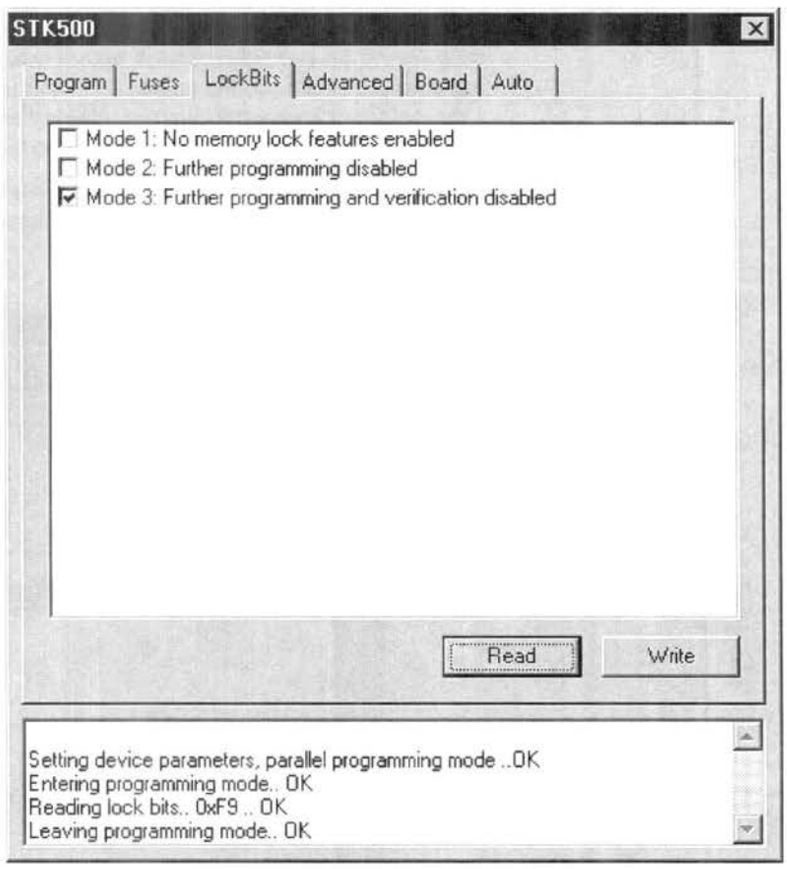
Figure 5-3. Fuses



5.3.3 "LockBits" Settings

Similar to the "Fuses" tab, the "LockBits" tab shows which lock modes are applicable to the selected device. All lock bits are accessible in both ISP and High-voltage Programming. A lock mode may consist of a combination of setting multiple Lock bits. This is handled by the STK500 user interface, and the correct lock bits are programmed automatically for the selected lock mode. Once a Lock mode protection level is enabled, it is not possible to lower the protection level by selecting a "lower" degree of protection or by setting a different Lock mode. The only way to remove a programmed Lock bit is to perform a complete chip erase, erasing both program and data memories. One exception exists: If the target device has a programmed "EESAVE" fuse, the contents of the EEPROM will be saved even though a complete chip erase on the device is performed.

Figure 5-4. LockBits



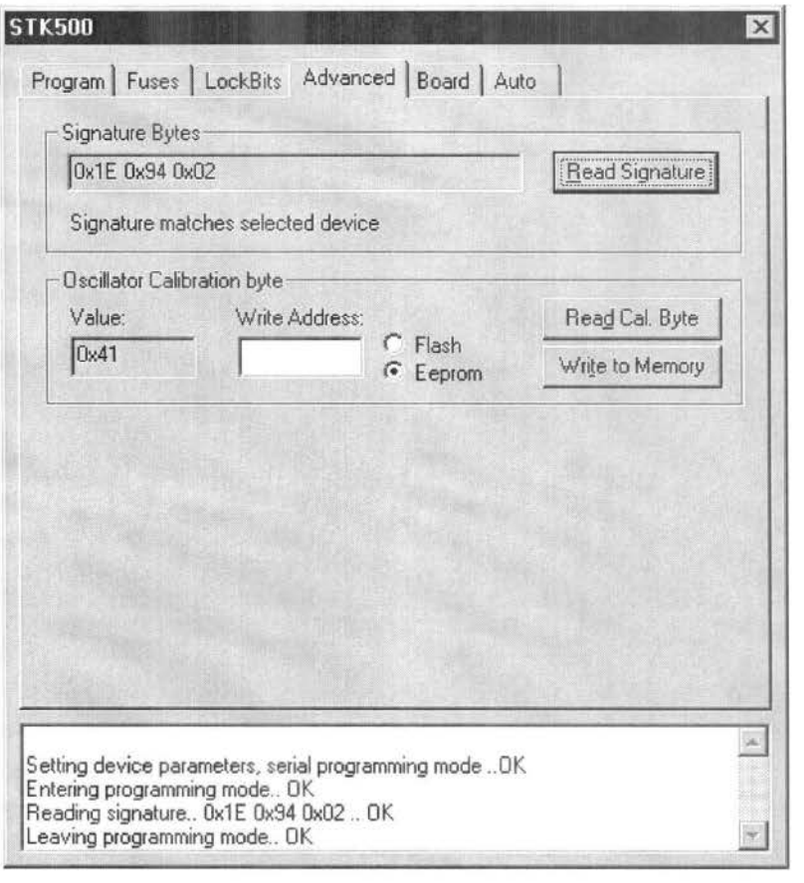
5.3.4 "Advanced" Settings

The "Advanced" tab is currently divided into two subgroups.

5.3.4.1 Signature Bytes

By pressing the "Read Signature" button, the signature bytes are read from the target device. The signature bytes act like an identifier for the part. After reading the signature, the software will check if it is the correct signature according to the chosen device. Please refer to the AVR datasheets to read more about signature bytes.

Figure 5-5. Advanced



5.3.4.2 Oscillator Calibration Byte

The oscillator calibration byte is written to the device during manufacturing, and cannot be erased or altered by the user. The calibration byte is a tuning value that should be written to the OSCCAL register in order to tune the internal RC oscillator.

5.3.4.3 Reading Oscillator Calibration Byte

By pressing the "Read Cal. Byte" button, the calibration value is read from the device and is shown in the "Value" text box. Note that the calibration byte is not directly accessible during program execution and must be written to a memory location during programming if it shall be used by the program. If this option is grayed out, the selected device does not have a tunable internal RC oscillator.

5.3.4.4 Writing Oscillator Calibration Byte

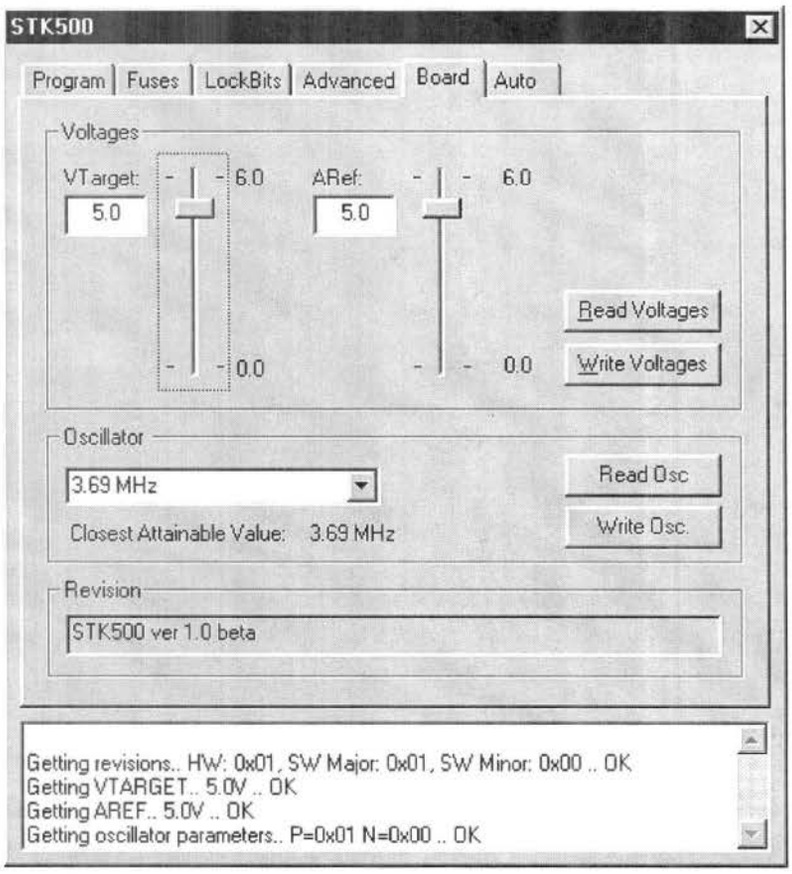
Since the calibration byte is not directly accessible during program execution, the user should write the calibration byte into a known location in Flash or EEPROM memory. Do this by writing the desired memory address in the "Write Address" text box and then press the "Write to Memory" button. The calibration byte is then written to the memory indicated by the "Flash" and "Eeprom" radio buttons.

5.3.5 "Board" Settings

The "Board" tab allows the changing of operating conditions on the STK500 development board. The following properties can be modified: VTARGET, AREF and oscillator frequency.

The interface is very flexible and it is possible to force the operating conditions beyond the recommended specifications for the device. Doing this is not recommended, and may damage the target device. The recommended operating conditions for the part are stated in the device datasheet.

Figure 5-6. Board



5.3.5.1 VTARGET

VTARGET controls the operating voltage for the target board. Through the use of the slide bar or the text box, this voltage can be regulated between 0 and 6.0V in 0.1V increments. Please refer to the device datasheet to find the specified voltage range for the selected device. Both voltages are read by pressing the "Read Voltages" button, and written by pressing the "Write Voltages" button.

The physical connection of the VTARGET voltage is shown in Figure 3-22 on page 3-16.

5.3.5.2 AREF

AREF controls the analog reference voltage for the ADC converter. This setting only apply to devices with AD converter. Through the use of the slide bar or the text box, this voltage can be regulated between 0 and 6.0V in 0.1V increments. Please refer to the device datasheet to find the valid voltage range for the selected device. Both VTARGET and AREF are read by pressing the "Read Voltages" button, and written by pressing the "Write Voltages" button.

It is not possible to set AREF to a higher voltage than VTARGET because this will permanently damage the AVR.

The physical connection of the AREF voltage is shown in Figure 3-24 on page 3-18.

5.3.5.3 Oscillator

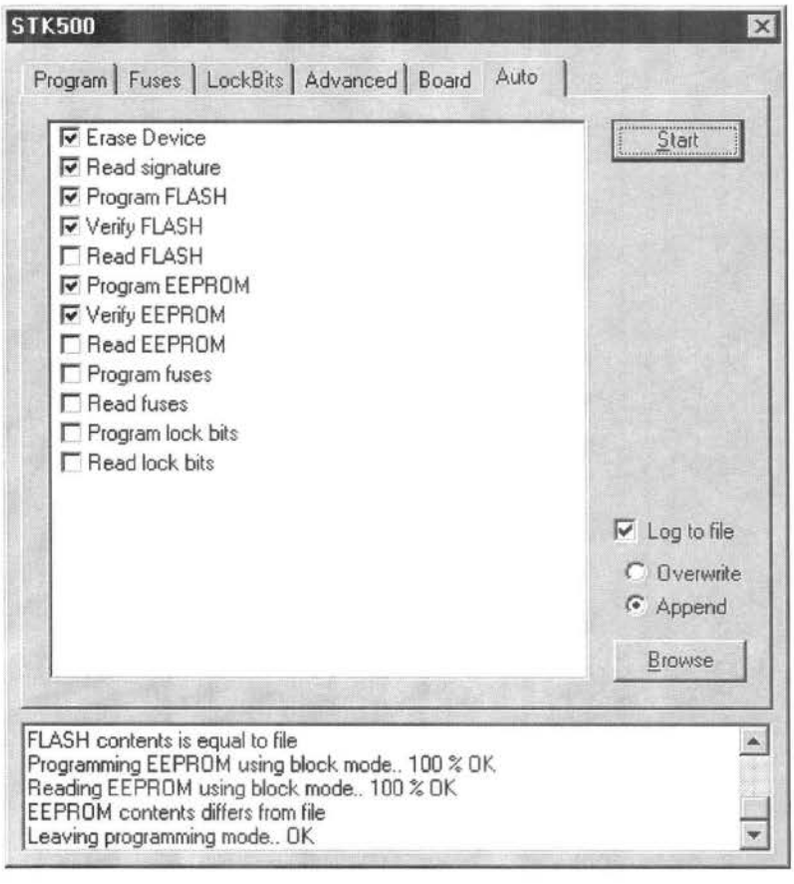
The STK500 development board uses a programmable oscillator circuit that offers a wide range of frequencies for the target device.

Since it is not possible to generate an unlimited number of frequencies, the STK500 user interface will calculate the value closest to the value written to the oscillator text box. The calculated value is then presented in the oscillator text box, overwriting the previously written number.

5.3.6 "Auto" Settings

When programming multiple devices with the same code, the "Auto" tab offers a powerful method of automatically going through a user-defined sequence of commands. The commands are listed in the order they are executed (if selected). To enable a command, the appropriate check box should be checked. For example, if only "Program FLASH" is checked when the "Start" button is pressed, the Flash memory will be programmed with the hex file specified in the "Program" settings. All commands depend on and use the settings given in the STK500 user interface.

Figure 5-7. Auto



It is possible to log the command execution to a text file by checking the "Log to file" check box.

5.3.6.1 Setting Up the System for Auto-programming

Click on the check boxes for the commands that you want the STK500 user interface to perform. A typical sequence where the device is erased and then programmed is shown in Figure 5-7. The chip is erased, both memories programmed and verified, and finally, fuses and lock bits are programmed.

Once configured, the same programming sequence is executed every time the "Start" button is pressed. This reduces both work and possibilities for errors due to operational errors.

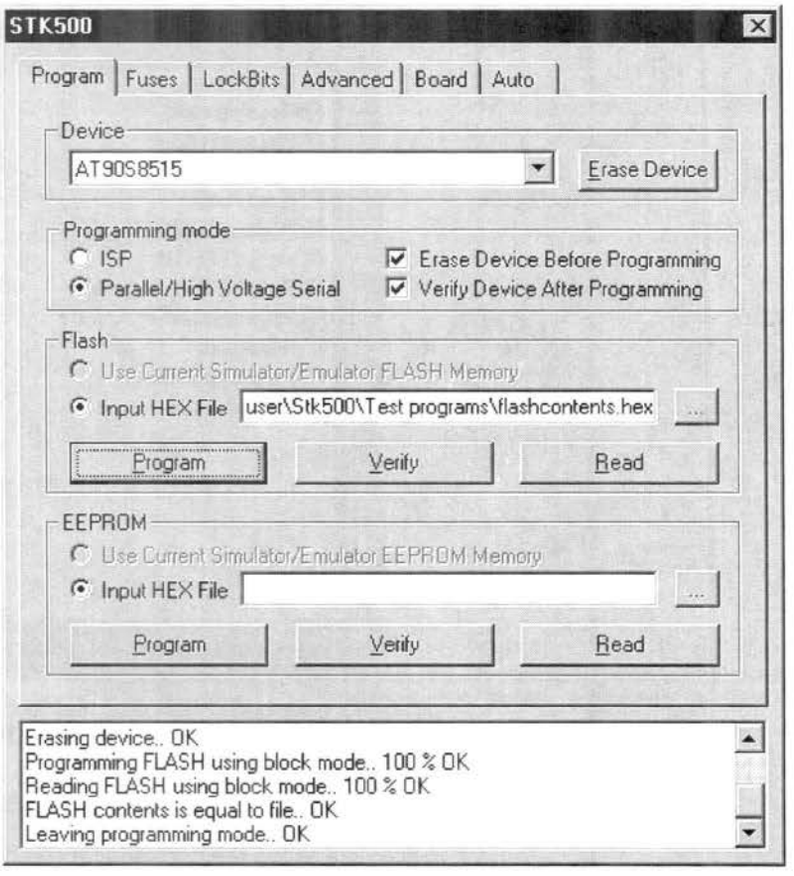
5.3.6.2 Logging the Auto-programming to a File

By clicking on the "Log to file" check box, all output from the commands are written to a text file. Select or create the file by pressing the "Browse" button and navigate to the location where the file is placed or should be created. The output is directed to this file, and can be viewed and edited using a text editor.

5.3.7 History Window

The History window is located at the bottom of the STK500 view. In this window the dialog between AVR Studio and STK500 is shown. For every new command performed, the old dialog is replaced with the new one.

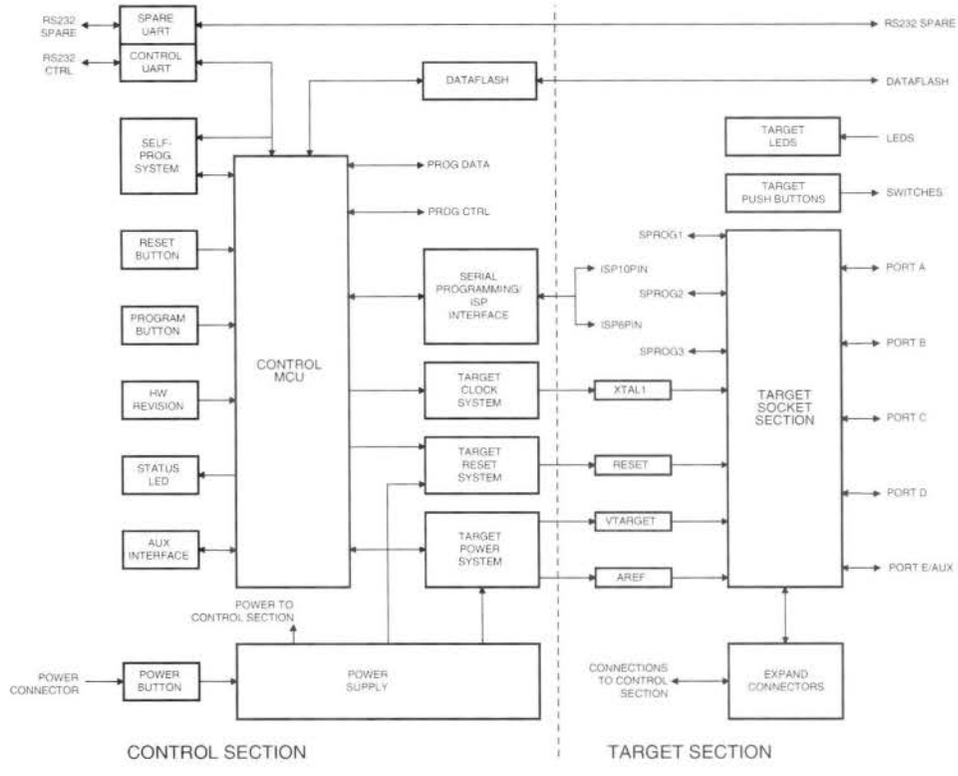
Figure 5-8. History Window



Section 10

Appendix A

Figure 10-1. STK500 Block Diagram



ΠΑΡΑΡΤΗΜΑ Γ

Στο παράρτημα αυτό περιλαμβάνονται ενδεικτικά βασικές πληροφορίες από το εγχειρίδιο της ηλεκτρονικής βιβλιοθήκης LIB-C, του περιβάλλοντος ανάπτυξης WINAVR. Καθώς και η περιγραφή των μη στάνταρ εντολών της C που χρησιμοποιήθηκαν στο πρόγραμμα

MAIN PAGE

1.4.6

Introduction

The latest version of this document is always available from
http://savannah.nongnu.org/projects/avr-libc/

The AVR Libc package provides a subset of the standard C library for Atmel AVR 8-bit RISC microcontrollers. In addition, the library provides the basic startup code needed by most applications.

There is a wealth of information in this document which goes beyond simply describing the interfaces and routines provided by the library. We hope that this document provides enough information to get a new AVR developer up to speed quickly using the freely available development tools: binutils, gcc avr-libc and many others.

If you find yourself stuck on a problem which this document doesn't quite address, you may wish to post a message to the avr-gcc mailing list. Most of the developers of the AVR binutils and gcc ports in addition to the devleopers of avr-libc subscribe to the list, so you will usually be able to get your problem resolved. You can subscribe to the list at http://lists.nongnu.org/mailman/listinfo/avr-gcc-list . Before posting to the list, you might want to try reading the Frequently Asked Questions chapter of this document.

Note:

If you think you've found a bug, or have a suggestion for an improvement, either in this documentation or in the library itself, please use the bug tracker at https://savannah.nongnu.org/bugs/?group=avr-libc to ensure the issue won't be forgotten.

General information about this library

In general, it has been the goal to stick as best as possible to established standards while implementing this library. Commonly, this refers to the C library as described by the ANSI X3.159-1989 and ISO/IEC 9899:1990 ("ANSI-C") standard, as well as parts of their successor ISO/IEC 9899:1999 ("C99"). Some additions have been inspired by other standards like IEEE Std 1003.1-1988 ("POSIX.1"), while other extensions are purely AVR-specific (like the entire program-space string interface).

Unless otherwise noted, functions of this library are *not* guarenteed to be reentrant. In particular, any functions that store local state are known to be non-reentrant, as well as functions that manipulate IO registers like the EEPROM access routines. If these

functions are used within both, standard and interrupt context, undefined behaviour will result.

Supported Devices

The following is a list of AVR devices currently supported by the library. Note that actual support for some newer devices depends on the ability of the compiler/assembler to support these devices at library compile-time.

AT90S Type Devices:

- at90s1200 [1]
- at90s2313
- at90s2323
- at90s2333
- at90s2343
- at90s4414
- at90s4433
- at90s4434
- at90s8515
- at90c8534
- at90s8535

ATmega Type Devices:

- atmega8
- atmega103
- atmega128
- atmega1280
- atmega1281
- atmega16
- atmega161
- atmega162
- atmega163
- atmega164p
- atmega165
- atmega165p
- atmega168
- atmega169
- atmega169p
- atmega2560
- atmega2561
- atmega32
- atmega323
- atmega324p
- atmega325

- atmega325p
- atmega3250
- atmega3250p
- atmega329
- atmega329p
- atmega3290
- atmega3290p
- atmega48
- atmega64
- atmega640
- atmega644
- atmega644p
- atmega645
- atmega6450
- atmega649
- atmega6490
- atmega8515
- atmega8535
- atmega88

ATtiny Type Devices:

- attiny11 [1]
- attiny12 [1]
- attiny13
- attiny15 [1]
- attiny22
- attiny24
- attiny25
- attiny26
- attiny261
- attiny28 [1]
- attiny2313
- attiny44
- attiny45
- attiny461
- attiny84
- attiny85
- attiny861

Misc Devices:

- at94K [2]
- at76c711 [3]
- at43usb320
- at43usb355

- at86rf401
- at90can32
- at90can64
- at90can128
- at90pwm1
- at90pwm2
- at90pwm3
- at90usb82
- at90usb162
- at90usb646
- at90usb647
- at90usb1286
- at90usb1287
- atmega8hva
- atmega16hva
- atmega406

Note:

[1] Assembly only. There is no direct support for these devices to be programmed in C since they do not have a RAM based stack. Still, it could be possible to program them in C, see the FAQ for an option.

Note:

[2] The at94K devices are a combination of FPGA and AVR microcontroller. [TRoth-2002/11/12: Not sure of the level of support for these. More information would be welcomed.]

Note:

[3] The at76c711 is a USB to fast serial interface bridge chip using an AVR core.

avr-libc License

avr-libc can be freely used and redistributed, provided the following license conditions are met.

Toolchain Overview

Introduction

Welcome to the open source software development toolset for the Atmel AVR!

There is not a single tool that provides everything needed to develop software for the AVR. It takes many tools working together. Collectively, the group of tools are called a toolset, or commonly a toolchain, as the tools are chained together to produce the final executable application for the AVR microcontroller.

The following sections provide an overview of all of these tools. You may be used to cross-compilers that provide everything with a GUI front-end, and not know what goes on "underneath the hood". You may be coming from a desktop or server computer background and not used to embedded systems. Or you may be just learning about the most common software development toolchain available on Unix and Linux systems. Hopefully the following overview will be helpful in putting everything in perspective.

FSF and GNU

According to its website, "the Free Software Foundation (FSF), established in 1985, is dedicated to promoting computer users' rights to use, study, copy, modify, and redistribute computer programs. The FSF promotes the development and use of free software, particularly the GNU operating system, used widely in its GNU/Linux variant." The FSF remains the primary sponsor of the GNU project.

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced guh-noo, approximately like canoe.

One of the main projects of the GNU system is the GNU Compiler Collection, or GCC, and its sister project, GNU Binutils. These two open source projects provide a foundation for a software development toolchain. Note that these projects were designed to originally run on Unix-like systems.

GCC

GCC stands for GNU Compiler Collection. GCC is highly flexible compiler system. It has different compiler front-ends for different languages. It has many back-ends that generate assembly code for many different processors and host operating systems. All share a common "middle-end", containing the generic parts of the compiler, including a lot of optimizations.

In GCC, a `host` system is the system (processor/OS) that the compiler runs on. A `target` system is the system that the compiler compiles code for. And, a system is the system that the compiler is built (from source code) on. If a compiler has the same system for `host` and for `target`, it is known as a `native` compiler. If a compiler has different systems for `host` and `target`, it is known as a cross-compiler. (And if all three, `build`, `host`, and `target` systems are different, it is known as a Canadian cross compiler, but we won't discuss that here.) When GCC is built to execute on a `host` system such as FreeBSD, Linux, or Windows, and it is built to generate code for the AVR microcontroller `target`, then it is a cross compiler, and this version of GCC is commonly known as "AVR GCC". In documentation, or discussion, AVR GCC is used when referring to GCC targeting specifically the AVR, or something that is AVR specific about GCC. The term "GCC" is usually used to refer to something generic about GCC, or about GCC as a whole.

GCC is different from most other compilers. GCC focuses on translating a high-level language to the target assembly only. AVR GCC has three available compilers for the AVR: C language, C++, and Ada. The compiler itself does not assemble or link the final code.

GCC is also known as a "driver" program, in that it knows about, and drives other programs seamlessly to create the final output. The assembler, and the linker are part of another open source project called GNU Binutils. GCC knows how to drive the GNU assembler (gas) to assemble the output of the compiler. GCC knows how to drive the GNU linker (ld) to link all of the object modules into a final executable.

The two projects, GCC and Binutils, are very much interrelated and many of the same volunteers work on both open source projects.

When GCC is built for the AVR target, the actual program names are prefixed with "avr-". So the actual executable name for AVR GCC is: avr-gcc. The name "avr-gcc" is used in documentation and discussion when referring to the program itself and not just the whole AVR GCC system.

See the GCC Web Site and GCC User Manual for more information about GCC.

GNU Binutils

The name GNU Binutils stands for "Binary Utilities". It contains the GNU assembler (gas), and the GNU linker (ld), but also contains many other utilities that work with binary files that are created as part of the software development toolchain.

Again, when these tools are built for the AVR target, the actual program names are prefixed with "avr-". For example, the assembler program name, for a native assembler is "as" (even though in documentation the GNU assembler is commonly referred to as "gas"). But when built for an AVR target, it becomes "avr-as". Below is a list of the programs that are included in Binutils:

avr-as
: The Assembler.

avr-ld
: The Linker.

avr-ar
: Create, modify, and extract from libraries (archives).

avr-ranlib
: Generate index to library (archive) contents.

avr-objcopy
: Copy and translate object files to different formats.

avr-objdump
: Display information from object files including disassembly.

avr-size
: List section sizes and total size.

avr-nm
: List symbols from object files.

avr-strings
: List printable strings from files.

avr-strip
: Discard symbols from files.

avr-readelf
: Display the contents of ELF format files.

avr-addr2line
: Convert addresses to file and line.

avr-c++filt
: Filter to demangle encoded C++ symbols.

avr-libc

GCC and Binutils provides a lot of the tools to develop software, but there is one critical component that they do not provide: a Standard C Library.

There are different open source projects that provide a Standard C Library depending upon your system time, whether for a native compiler (GNU Libc), for some other embedded system (newlib), or for some versions of Linux (uCLibc). The open source AVR toolchain has its own Standard C Library project: avr-libc.

AVR-Libc provides many of the same functions found in a regular Standard C Library and many additional library functions that is specific to an AVR. Some of the Standard C Library functions that are commonly used on a PC environment have limitations or additional issues that a user needs to be aware of when used on an embedded system.

AVR-Libc also contains the most documentation about the whole AVR toolchain.

Building Software

Even though GCC, Binutils, and avr-libc are the core projects that are used to build software for the AVR, there is another piece of software that ties it all together: Make. GNU Make is a program that makes things, and mainly software. Make interprets and executes a Makefile that is written for a project. A Makefile contains dependency rules, showing which output files are dependent upon which input files, and instructions on how to build output files from input files.

Some distributions of the toolchains, and other AVR tools such as MFile, contain a Makefile template written for the AVR toolchain and AVR applications that you can copy and modify for your application.

See the GNU Make User Manual for more information.

AVRDUDE

After creating your software, you'll want to program your device. You can do this by using the program AVRDUDE which can interface with various hardware devices to program your processor.

AVRDUDE is a very flexible package. All the information about AVR processors and various hardware programmers is stored in a text database. This database can be modified by any user to add new hardware or to add an AVR processor if it is not already listed.

GDB / Insight / DDD

The GNU Debugger (GDB) is a command-line debugger that can be used with the rest of the AVR toolchain. Insight is GDB plus a GUI written in Tcl/Tk. Both GDB and Insight are configured for the AVR and the main executables are prefixed with the target name: avr-gdb, and avr-insight. There is also a "text mode" GUI for GDB: avr-gdbtui. DDD (Data Display Debugger) is another popular GUI front end to GDB, available on Unix and Linux systems.

AVaRICE

AVaRICE is a back-end program to AVR GDB and interfaces to the Atmel JTAG In-Circuit Emulator (ICE), to provide emulation capabilities.

SimulAVR

SimulAVR is an AVR simulator used as a back-end with AVR GDB. Unfortunately, this project is currently unmaintained and could use some help.

Utilities

There are also other optional utilities available that may be useful to add to your toolset.

`SRecord` is a collection of powerful tools for manipulating EPROM load files. It reads and writes numerous EPROM file formats, and can perform many different manipulations.

`MFile` is a simple Makefile generator is meant as an aid to quickly customize a Makefile to use for your AVR application.

Toolchain Distributions (Distros)

All of the various open source projects that comprise the entire toolchain are normally distributed as source code. It is left up to the user to build the tool application from its source code. This can be a very daunting task to any potential user of these tools.

Luckily there are people who help out in this area. Volunteers take the time to build the application from source code on particular host platforms and sometimes packaging the tools for convenient installation by the end user. These packages contain the binary executables of the tools, pre-made and ready to use. These packages are known as "distributions" of the AVR toolchain, or by a more shortened name, "distros".

AVR toolchain distros are available on FreeBSD, Windows, Mac OS X, and certain flavors of Linux.

Open Source

All of these tools, from the original source code in the multitude of projects, to the various distros, are put together by many, many volunteers. All of these projects could always use more help from other people who are willing to volunteer some of their time. There are many different ways to help, for people with varying skill levels, abilities, and available time.

You can help to answer questions in mailing lists such as the avr-gcc-list, or on forums at the AVR Freaks website. This helps many people new to the open source AVR tools.

If you think you found a bug in any of the tools, it is always a big help to submit a good bug report to the proper project. A good bug report always helps other volunteers to analyze the problem and to get it fixed for future versions of the software.

You can also help to fix bugs in various software projects, or to add desirable new features.

Volunteers are always welcome! :-)

LIBRARY REFERENCE

Here is a list of all modules:

Automatically generated by Doxygen 1.4.7 on 15 May 2007.

<stdint.h>: Standard Integer Types

Detailed Description

```
#include <stdint.h>
```

Use [u]intN_t if you need exactly N bits.

Since these typedefs are mandated by the C99 standard, they are preferred over rolling your own typedefs.

Typedef Documentation

typedef unsigned char uint8_t

8-bit unsigned type.

<avr/interrupt.h>: Interrupts

Detailed Description

Note:

This discussion of interrupts was originally taken from Rich Neswold's document. See Acknowledgments.

Introduction to avr-libc's interrupt handling

It's nearly impossible to find compilers that agree on how to handle interrupt code. Since the C language tries to stay away from machine dependent details, each compiler writer is forced to design their method of support.

In the AVR-GCC environment, the vector table is predefined to point to interrupt routines with predetermined names. By using the appropriate name, your routine will be called when the corresponding interrupt occurs. The device library provides a set of default interrupt routines, which will get used if you don't define your own.

Patching into the vector table is only one part of the problem. The compiler uses, by convention, a set of registers when it's normally executing compiler-generated code. It's important that these registers, as well as the status register, get saved and restored. The extra code needed to do this is enabled by tagging the interrupt function with `__attribute__((signal))`.

These details seem to make interrupt routines a little messy, but all these details are handled by the Interrupt API. An interrupt routine is defined with ISR(). This macro register and mark the routine as an interrupt handler for the specified peripheral. The following is an example definition of a handler for the ADC interrupt.

```
#include <avr/interrupt.h>

ISR(ADC_vect)
{
    // user code here
}
```

Refer to the chapter explaining assembler programming for an explanation about interrupt routines written solely in assembler language.

Catch-all interrupt vector

If an unexpected interrupt occurs (interrupt is enabled and no handler is installed, which usually indicates a bug), then the default action is to reset the device by jumping to the

reset vector. You can override this by supplying a function named __vector_default which should be defined with ISR() as such.

```
#include <avr/interrupt.h>

ISR(__vector_default)
{
    // user code here
}
```

Nested interrupts

The AVR hardware clears the global interrupt flag in SREG before entering an interrupt vector. Thus, normally interrupts will remain disabled inside the handler until the handler exits, where the RETI instruction (that is emitted by the compiler as part of the normal function epilogue for an interrupt handler) will eventually re-enable further interrupts. For that reason, interrupt handlers normally do not nest. For most interrupt handlers, this is the desired behaviour, for some it is even required in order to prevent infinitely recursive interrupts (like UART interrupts, or level-triggered external interrupts). In rare circumstances though it might be desired to re-enable the global interrupt flag as early as possible in the interrupt handler, in order to not defer any other interrupt more than absolutely needed. This could be done using an sei() instruction right at the beginning of the interrupt handler, but this still leaves few instructions inside the compiler-generated function prologue to run with global interrupts disabled. The compiler can be instructed to insert an SEI instruction right at the beginning of an interrupt handler by declaring the handler the following way:

```
void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
  ...
}
```

where XXX_vect is the name of a valid interrupt vector for the MCU type in question, as explained below.

Choosing the vector: Interrupt vector names

The interrupt is chosen by supplying one of the symbols in following table.

There are currently two different styles present for naming the vectors. One form uses names starting with SIG_, followed by a relatively verbose but arbitrarily chosen name describing the interrupt vector. This has been the only available style in avr-libc up to version 1.2.x.

Starting with avr-libc version 1.4.0, a second style of interrupt vector names has been added, where a short phrase for the vector description is followed by _vect. The short phrase matches the vector name as described in the datasheet of the respective device (and in Atmel's XML files), with spaces replaced by an underscore and other non-

alphanumeric characters dropped. Using the suffix `_vect` is intented to improve portability to other C compilers available for the AVR that use a similar naming convention.

The historical naming style might become deprecated in a future release, so it is not recommended for new projects.

Note:

The ISR() macro cannot really spell-check the argument passed to them. Thus, by misspelling one of the names below in a call to ISR(), a function will be created that, while possibly being usable as an interrupt function, is not actually wired into the interrupt vector table. The compiler will generate a warning if it detects a suspiciously looking name of a ISR() function (i.e. one that after macro replacement does not start with "__vector_").

(OMITED TABLE WITH COMPLETE LIST OF AVR INTERRUPTS AND APPLICABLE CEVICES)

Global manipulation of the interrupt flag

The global interrupt flag is maintained in the I bit of the status register (SREG).

#define sei()
#define cli()

Macros for writing interrupt handler functions

#define ISR(vector)
#define SIGNAL(vector)
#define EMPTY_INTERRUPT(vector)
#define ISR_ALIAS(vector, target_vector)

Define Documentation

#define cli ( )

```
#include <avr/interrupt.h>
```

Disables all interrupts by clearing the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead.

```
#define ISR ( vector )

#include <avr/interrupt.h>
```

Introduces an interrupt handler function (interrupt service routine) that runs with global interrupts initially disabled.

`vector` must be one of the interrupt vector names that are valid for the particular MCU type.

#define sei ( )

```
#include <avr/interrupt.h>
```

Enables interrupts by setting the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead.

<avr/sfr_defs.h>: Special function registers

Detailed Description

When working with microcontrollers, many of the tasks usually consist of controlling the peripherals that are connected to the device, respectively programming the subsystems that are contained in the controller (which by itself communicate with the circuitry connected to the controller).

The AVR series of microcontrollers offers two different paradigms to perform this task. There's a separate IO address space available (as it is known from some high-level CISC CPUs) that can be addressed with specific IO instructions that are applicable to some or all of the IO address space (`in`, `out`, `sbi` etc.). The entire IO address space is also made available as *memory-mapped IO*, i. e. it can be accessed using all the MCU instructions that are applicable to normal data memory. The IO register space is mapped into the data memory address space with an offset of 0x20 since the bottom of this space is reserved for direct access to the MCU registers. (Actual SRAM is available only behind the IO register area, starting at either address 0x60, or 0x100 depending on the device.)

AVR Libc supports both these paradigms. While by default, the implementation uses memory-mapped IO access, this is hidden from the programmer. So the programmer can access IO registers either with a special function like `outb()`:

```
#include <avr/io.h>

outb(PORTA, 0x33);
```

or they can assign a value directly to the symbolic address:

```
PORTA = 0x33;
```

The compiler's choice of which method to use when actually accessing the IO port is completely independent of the way the programmer chooses to write the code. So even if the programmer uses the memory-mapped paradigm and writes

```
PORTA |= 0x40;
```

the compiler can optimize this into the use of an `sbi` instruction (of course, provided the target address is within the allowable range for this instruction, and the right-hand side of the expression is a constant value known at compile-time).

The advantage of using the memory-mapped paradigm in C programs is that it makes the programs more portable to other C compilers for the AVR platform. Some people might also feel that this is more readable. For example, the following two statements would be equivalent:

```
outb(DDRD, inb(DDRD) & ~LCDBITS);
DDRD &= ~LCDBITS;
```

The generated code is identical for both. Without optimization, the compiler strictly generates code following the memory-mapped paradigm, while with optimization turned on, code is generated using the (faster and smaller) `in/out` MCU instructions.

Note that special care must be taken when accessing some of the 16-bit timer IO registers where access from both the main program and within an interrupt context can happen. See Why do some 16-bit timer registers sometimes get trashed?.

Porting programs that use sbi/cbi
As described above, access to the AVR single bit set and clear instructions are provided via the standard C bit manipulation commands. The sbi and cbi commands are no longer directly supported. sbi (sfr,bit) can be replaced by sfr |= _BV(bit) .

ie: sbi(PORTB, PB1); is now PORTB |= _BV(PB1);

This actually is more flexible than having sbi directly, as the optimizer will use a hardware sbi if appropriate, or a read/or/write if not. You do not need to keep track of which registers sbi/cbi will operate on.

Likewise, cbi (sfr,bit) is now sfr &= ~(_BV(bit));

Modules

Additional notes from <avr/sfr_defs.h>

Bit manipulation

#define _BV(bit) (1 << (bit))

IO register bit manipulation

#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
#define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
#define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))

Define Documentation

#define loop_until_bit_is_clear (sfr, bit) do { } while (bit_is_set(sfr, bit))

#include <avr/io.h>

Wait until bit bit in IO register sfr is clear.

#define loop_until_bit_is_set (sfr, bit) do { } while (bit_is_clear(sfr, bit))

#include <avr/io.h>

Wait until bit bit in IO register sfr is set.

<util/delay.h>: Convenience functions for busy-wait delay loops

Detailed Description

```
#define F_CPU 1000000UL  // 1 MHz
//#define F_CPU 14.7456E6
#include <util/delay.h>
```

Note:

As an alternative method, it is possible to pass the F_CPU macro down to the compiler from the Makefile. Obviously, in that case, no `#define` statement should be used.

The functions in this header file are wrappers around the basic busy-wait functions from <util/delay_basic.h>. They are meant as convenience functions where actual time values can be specified rather than a number of cycles to wait for. The idea behind is that compile-time constant expressions will be eliminated by compiler optimization so floating-point expressions can be used to calculate the number of delay cycles needed based on the CPU frequency passed by the macro F_CPU.

Note:

In order for these functions to work as intended, compiler optimizations *must* be enabled, and the delay time *must* be an expression that is a known constant at compile-time. If these requirements are not met, the resulting delay will be much longer (and basically unpredictable), and applications that otherwise do not use floating-point calculations will experience severe code bloat by the floating-point library routines linked into the application.

The functions available allow the specification of microsecond, and millisecond delays directly, using the application-supplied macro F_CPU as the CPU clock frequency (in Hertz).

Functions

void _delay_us (double __us)
void _delay_ms (double __ms)

Function Documentation

void _delay_ms (double__ms)

Perform a delay of __ms milliseconds, using _delay_loop_2().

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency (in Hertz).

The maximal possible delay is 262.14 ms / F_CPU in MHz.

void _delay_us (double__us)

Perform a delay of __us microseconds, using _delay_loop_1().

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency (in Hertz).

The maximal possible delay is 768 us / F_CPU in MHz.

<util/parity.h>: Parity bit generation

Detailed Description

```
#include <util/parity.h>
```

This header file contains optimized assembler code to calculate the parity bit for a byte.

Defines

#define parity_even_bit(val)

Define Documentation

#define parity_even_bit(val)

Value:

```
(__extension__({                                        \
        unsigned char __t;                              \
        __asm__ (                                       \
                "mov __tmp_reg__,%0" "\n\t"             \
                "swap %0" "\n\t"                        \
                "eor %0,__tmp_reg__" "\n\t"             \
                "mov __tmp_reg__,%0" "\n\t"             \
                "lsr %0" "\n\t"                         \
                "lsr %0" "\n\t"                         \
                "eor %0,__tmp_reg__"                    \
                : "=r" (__t)                            \
                : "0" ((unsigned char)(val))            \
                : "r0"                                  \
        );                                              \
        (((__t + 1) >> 1) & 1);                         \
 }))
```

Returns:

1 if `val` has an odd number of bits set.

<math.h>: Mathematics

Detailed Description

```
#include <math.h>
```

This header file declares basic mathematics constants and functions.

Note:

In order to access the functions delcared herein, it is usually also required to additionally link against the library `libm.a`. See also the related FAQ entry.

Defines

#define M_PI 3.14159265358979323846264 3

#define M_SQRT2 1.41421356237309504880168872

Functions

double cos (double __x) __ATTR_CONST__
double fabs (double __x) __ATTR_CONST__
double fmod (double __x, double __y) __ATTR_CONST__
double modf (double __value, double *__iptr)
double sin (double __x) __ATTR_CONST__
double sqrt (double __x) __ATTR_CONST__
double tan (double __x) __ATTR_CONST__
double floor (double __x) __ATTR_CONST__
double ceil (double __x) __ATTR_CONST__
double frexp (double __value, int *__exp)
double ldexp (double __x, int __exp) __ATTR_CONST__
double exp (double __x) __ATTR_CONST__
double cosh (double __x) __ATTR_CONST__
double sinh (double __x) __ATTR_CONST__
double tanh (double __x) __ATTR_CONST__
double acos (double __x) __ATTR_CONST__
double asin (double __x) __ATTR_CONST__
double atan (double __x) __ATTR_CONST__
double atan2 (double __y, double __x) __ATTR_CONST__
double log (double __x) __ATTR_CONST__
double log10 (double __x) __ATTR_CONST__
double pow (double __x, double __y) __ATTR_CONST__

int isnan (double __x) __ATTR_CONST__
int isinf (double __x) __ATTR_CONST__
double square (double __x) __ATTR_CONST__

Define Documentation

#define M_PI 3.14159265358979323846264 3

The constant `pi`.

#define M_SQRT2 1.41421356237309504880168872

The square root of 2.

Function Documentation

double atan2 (double __y, double __x)

The atan2() function computes the principal value of the arc tangent of `y / x`, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-pi, +pi] radians. If both `x` and `y` are zero, the global variable `errno` is set to `EDOM`.

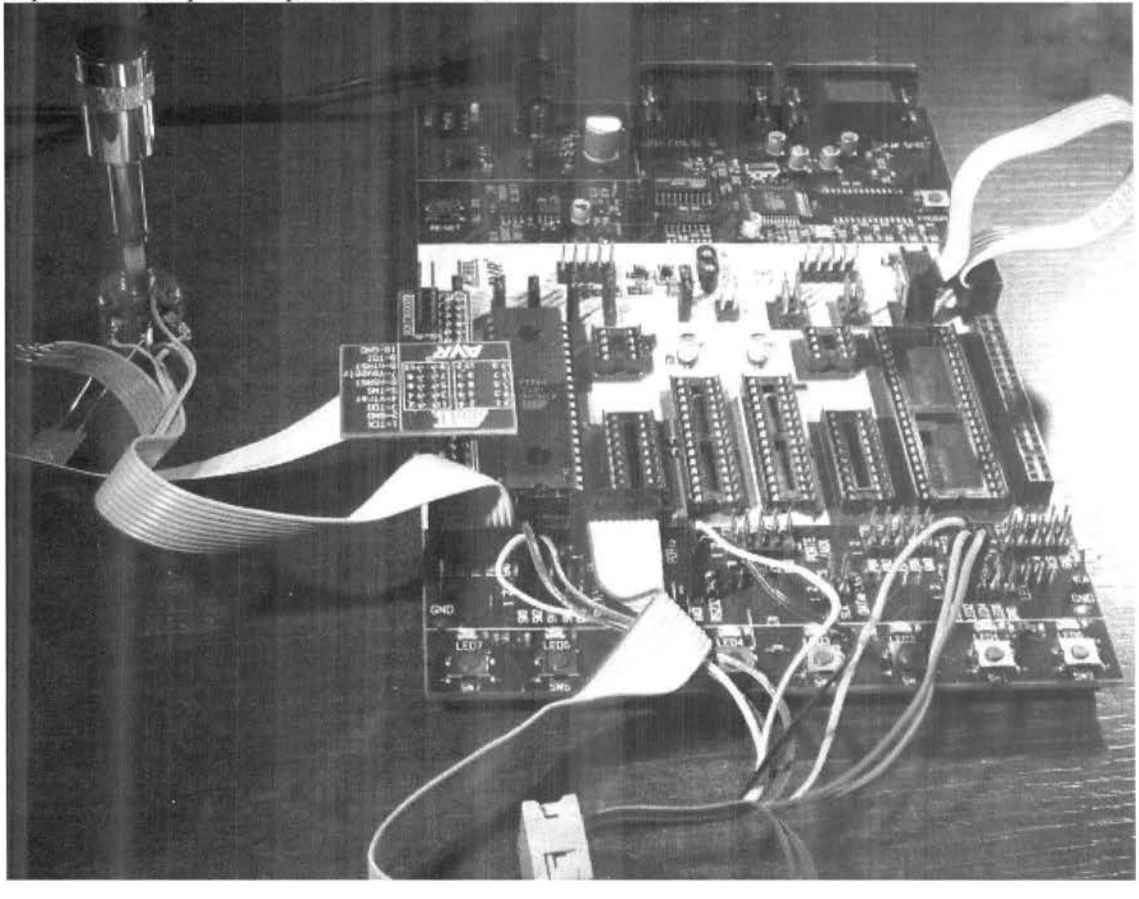
A more sophisticated project
[Demo projects]

This project extends the basic idea of the simple project to control a LED with a PWM output, but adds methods to adjust the LED brightness. It employs a lot of the basic concepts of avr-libc to achieve that goal.

Understanding this project assumes the simple project has been understood in full, as well as being acquainted with the basic hardware concepts of an AVR microcontroller.

Hardware setup

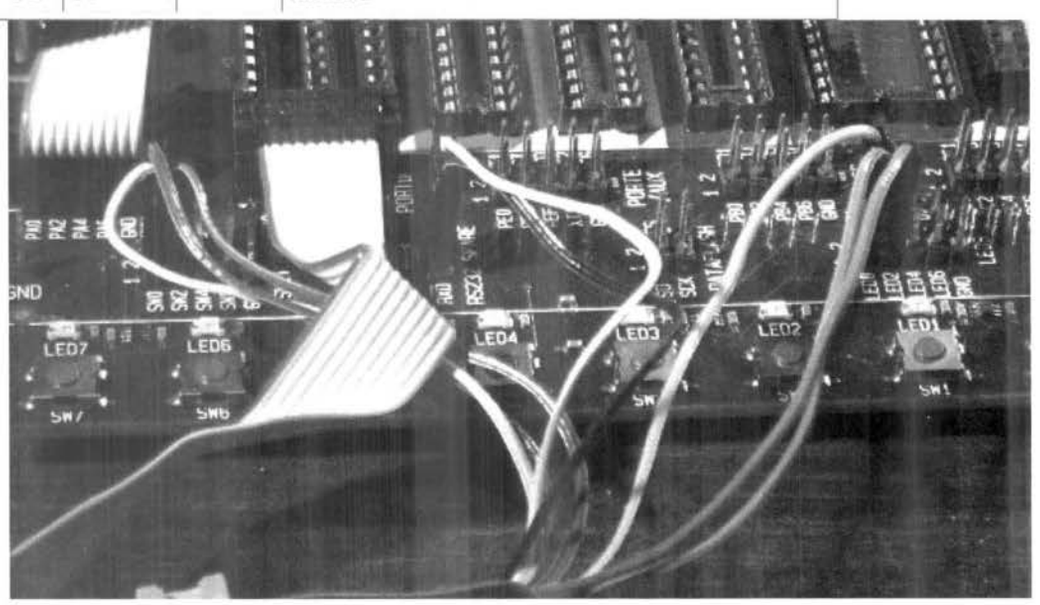
The demo is set up in a way so it can be run on the ATmega16 that ships with the STK500 development kit. The only external part needed is a potentiometer attached to the ADC. It is connected to a 10-pin ribbon cable for port A, both ends of the potentiometer to pins 9 (GND) and 10 (VCC), and the wiper to pin 1 (port A0). A bypass capacitor from pin 1 to pin 9 (like 47 nF) is recommendable.



Setup of the STK500

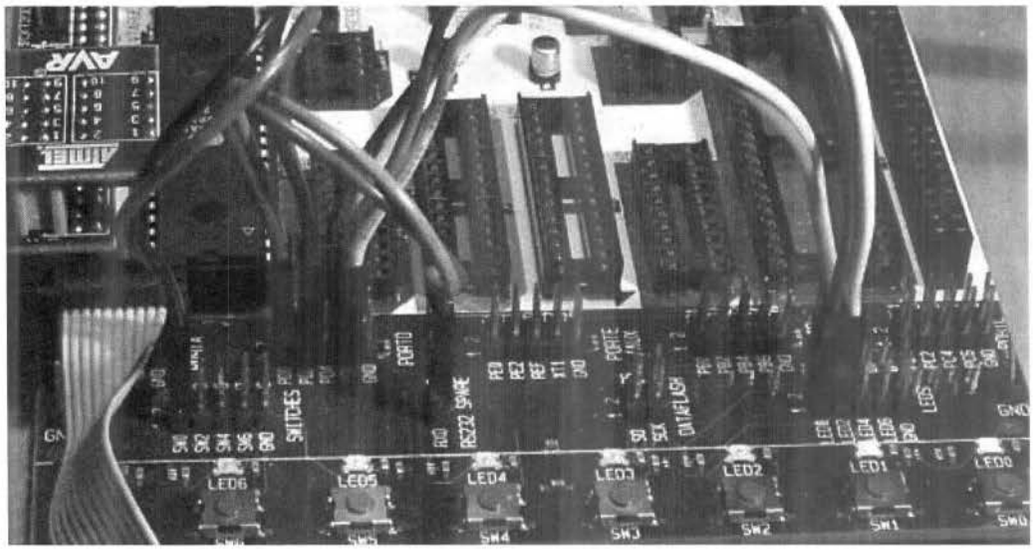
The coloured patch cables are used to provide various interconnections. As there are only four of them in the STK500, there are two options to connect them for this demo. The second option for the yellow-green cable is shown in parenthesis in the table. Alternatively, the "squid" cable from the JTAG ICE kit can be used if available.

Port	Header	Color	Function	Connect to
D0	1	brown	RxD	RXD of the RS-232 header
D1	2	grey	TxD	TXD of the RS-232 header
D2	3	black	button "down"	SW0 (pin 1 switches header)
D3	4	red	button "up"	SW1 (pin 2 switches header)
D4	5	green	button "ADC"	SW2 (pin 3 switches header)
D5	6	blue	LED	LED0 (pin 1 LEDs header)
D6	7	(green)	clock out	LED1 (pin 2 LEDs header)
D7	8	white	1-second flash	LED2 (pin 3 LEDs header)
GND	9		unused	
VCC	10		unused	



Wiring of the STK500

The following picture shows the alternate wiring where LED1 is connected but SW2 is not:



Wiring option #2 of the STK500

As an alternative, this demo can also be run on the popular ATmega8 controller, or its successor ATmega88 as well as the ATmega48 and ATmega168 variants of the latter. These controllers do not have a port named "A", so their ADC inputs are located on port C instead, thus the potentiometer needs to be attached to port C. Likewise, the OC1A output is not on port D pin 5 but on port B pin 1 (PB1). Thus, the above cabling scheme needs to be changed so that PB1 connects to the LED0 pin. (PD6 remains unconnected.) When using the STK500, use one of the jumper cables for this connection. All other port D pins should be connected the same way as described for the ATmega16 above.

When not using an STK500 starter kit, attach the LEDs through some resistor to Vcc (low-active LEDs), and attach pushbuttons from the respective input pins to GND. The internal pull-up resistors are enabled for the pushbutton pins, so no external resistors are needed.

Finally, the demo has been ported to the ATtiny2313 as well. As this AVR does not offer an ADC, everything related to handling the ADC is disabled in the code for that MCU type. Also, port D of this controller type only features 6 pins, so the 1-second flash LED had to be moved from PD6 to PD4. (PD4 is used as the ADC control button on the other MCU types, but that is not needed here.) OC1A is located at PB3 on this device.

The `MCU_TARGET` macro in the Makefile needs to be adjusted appropriately for the alternative controller types.

The flash ROM and RAM consumption of this demo are way below the resources of even an ATmega48, and still well within the capabilities of an ATtiny2313. The major advantage of experimenting with the ATmega16 (in addition that it ships together with an STK500 anyway) is that it can be debugged online via JTAG. Likewise, the

ATmega48/88/168 and ATtiny2313 devices can be debugged through debugWire, using the Atmel JTAG ICE mkII or the low-cost AVR Dragon.

Note that in the explanation below, all port/pin names are applicable to the ATmega16 setup.

Functional overview

PD6 will be toggled with each internal clock tick (approx. 10 ms). PD7 will flash once per second.

PD0 and PD1 are configured as UART IO, and can be used to connect the demo kit to a PC (9600 Bd, 8N1 frame format). The demo application talks to the serial port, and it can be controlled from the serial port.

PD2 through PD4 are configured as inputs, and control the application unless control has been taken over by the serial port. Shorting PD2 to GND will decrease the current PWM value, shorting PD3 to GND will increase it.

While PD4 is shorted to GND, one ADC conversion for channel 0 (ADC input is on PA0) will be triggered each internal clock tick, and the resulting value will be used as the PWM value. So the brightness of the LED follows the analog input value on PC0. VAREF on the STK500 should be set to the same value as VCC.

When running in serial control mode, the function of the watchdog timer can be demonstrated by typing an `r'. This will make the demo application run in a tight loop without retriggering the watchdog so after some seconds, the watchdog will reset the MCU. This situation can be figured out on startup by reading the MCUCSR register.

The current value of the PWM is backed up in an EEPROM cell after about 3 seconds of idle time after the last change. If that EEPROM cell contains a reasonable (i. e. non-erased) value at startup, it is taken as the initial value for the PWM. This virtually preserves the last value across power cycles. By not updating the EEPROM immmediately but only after a timeout, EEPROM wear is reduced considerably compared to immediately writing the value at each change.

A code walkthrough

This section explains the ideas behind individual parts of the code. The source code has been divided into numbered parts, and the following subsections explain each of these parts.

Part 1: Macro definitions

A number of preprocessor macros are defined to improve readability and/or portability of the application.

The first macros describe the IO pins our LEDs and pushbuttons are connected to. This provides some kind of mini-HAL (hardware abstraction layer) so should some of the connections be changed, they don't need to be changed inside the code but only on top. Note that the location of the PWM output itself is mandated by the hardware, so it cannot be easily changed. As the ATmega48/88/168 controllers belong to a more recent generation of AVRs, a number of register and bit names have been changed there, so they are mapped back to their ATmega8/16 equivalents to keep the actual program code portable.

The name `F_CPU` is the conventional name to describe the CPU clock frequency of the controller. This demo project just uses the internal calibrated 1 MHz RC oscillator that is enabled by default. Note that when using the `<util/delay.h>` functions, `F_CPU` needs to be defined before including that file.

The remaining macros have their own comments in the source code. The macro `TMR1_SCALE` shows how to use the preprocessor and the compiler's constant expression computation to calculate the value of timer 1's post-scaler in a way so it only depends on `F_CPU` and the desired software clock frequency. While the formula looks a bit complicated, using a macro offers the advantage that the application will automatically scale to new target softclock or master CPU frequencies without having to manually re-calculate hardcoded constants.

Part 2: Variable definitions

The `intflags` structure demonstrates a way to allocate bit variables in memory. Each of the interrupt service routines just sets one bit within that structure, and the application's main loop then monitors the bits in order to act appropriately.

Like all variables that are used to communicate values between an interrupt service routine and the main application, it is declared volatile.

The variable `ee_pwm` is not a variable in the classical C sense that could be used as an lvalue or within an expression to obtain its value. Instead, the

```
__attribute__((section(".eeprom")))
```

marks it as belonging to the EEPROM section. This section is merely used as a placeholder so the compiler can arrange for each individual variable's location in EEPROM. The compiler will also keep track of initial values assigned, and usually the Makefile is arranged to extract these initial values into a separate load file (`largedemo_eeprom.*` in this case) that can be used to initialize the EEPROM.

The actual EEPROM IO must be performed manually.

Similarly, the variable `mcucsr` is kept in the .noinit section in order to prevent it from being cleared upon application startup.

Part 3: Interrupt service routines

The ISR to handle timer 1's overflow interrupt arranges for the software clock. While timer 1 runs the PWM, it calls its overflow handler rather frequently, so the `TMR1_SCALE` value is used as a postscaler to reduce the internal software clock frequency further. If the software clock triggers, it sets the `tmr_int` bitfield, and defers all further tasks to the main loop.

The ADC ISR just fetches the value from the ADC conversion, disables the ADC interrupt again, and announces the presence of the new value in the `adc_int` bitfield. The interrupt is kept disabled while not needed, because the ADC will also be triggered by executing the SLEEP instruction in idle mode (which is the default sleep mode). Another option would be to turn off the ADC completely here, but that increases the ADC's startup time (not that it would matter much for this application).

Part 4: Auxiliary functions

The function `handle_mcucsr()` uses two `__attribute__` declarators to achieve specific goals. First, it will instruct the compiler to place the generated code into the .init3 section of the output. Thus, it will become part of the application initialization sequence. This is done in order to fetch (and clear) the reason of the last hardware reset from `MCUCSR` as early as possible. There is a short period of time where the next reset could already trigger before the current reason has been evaluated. This also explains why the variable `mcucsr` that mirrors the register's value needs to be placed into the .noinit section, because otherwise the default initialization (which happens after .init3) would blank the value again.

As the initialization code is not called using CALL/RET instructions but rather concatenated together, the compiler needs to be instructed to omit the entire function prologue and epilogue. This is performed by the *naked* attribute. So while syntactically, `handle_mcucsr()` is a function to the compiler, the compiler will just emit the instructions for it without setting up any stack frame, and not even a RET instruction at the end.

Function `ioinit()` centralizes all hardware setup. The very last part of that function demonstrates the use of the EEPROM variable `ee_pwm` to obtain an EEPROM address that can in turn be applied as an argument to `eeprom_read_word()`.

The following functions handle UART character and string output. (UART input is handled by an ISR.) There are two string output functions, `printstr()` and `printstr_p()`. The latter function fetches the string from program memory. Both functions translate a newline character into a carriage return/newline sequence, so a simple `\n` can be used in the source code.

The function `set_pwm()` propagates the new PWM value to the PWM, performing range checking. When the value has been changed, the new percentage will be announced on the serial link. The current value is mirrored in the variable `pwm` so others can use it in

calculations. In order to allow for a simple calculation of a percentage value without requiring floating-point mathematics, the maximal value of the PWM is restricted to 1000 rather than 1023, so a simple division by 10 can be used. Due to the nature of the human eye, the difference in LED brightness between 1000 and 1023 is not noticable anyway.

Part 5: main()

At the start of `main()`, a variable `mode` is declared to keep the current mode of operation. An enumeration is used to improve the readability. By default, the compiler would allocate a variable of type *int* for an enumeration. The *packed* attribute declarator instructs the compiler to use the smallest possible integer type (which would be an 8-bit type here).

After some initialization actions, the application's main loop follows. In an embedded application, this is normally an infinite loop as there is nothing an application could "exit" into anyway.

At the beginning of the loop, the watchdog timer will be retriggered. If that timer is not triggered for about 2 seconds, it will issue a hardware reset. Care needs to be taken that no code path blocks longer than this, or it needs to frequently perform watchdog resets of its own. An example of such a code path would be the string IO functions: for an overly large string to print (about 2000 characters at 9600 Bd), they might block for too long.

The loop itself then acts on the interrupt indication bitfields as appropriate, and will eventually put the CPU on sleep at its end to conserve power.

The first interrupt bit that is handled is the (software) timer, at a frequency of approximately 100 Hz. The `CLOCKOUT` pin will be toggled here, so e. g. an oscilloscope can be used on that pin to measure the accuracy of our software clock. Then, the LED flasher for LED2 ("We are alive"-LED) is built. It will flash that LED for about 50 ms, and pause it for another 950 ms. Various actions depending on the operation mode follow. Finally, the 3-second backup timer is implemented that will write the PWM value back to EEPROM once it is not changing anymore.

The ADC interrupt will just adjust the PWM value only.

Finally, the UART Rx interrupt will dispatch on the last character received from the UART.

All the string literals that are used as informational messages within `main()` are placed in program memory so no SRAM needs to be allocated for them. This is done by using the PSTR macro, and passing the string to `printstr_p()`.

The source code

Source file: largedemo.c

Frequently Asked Questions

My program doesn't recognize a variable updated within an interrupt routine

When using the optimizer, in a loop like the following one:

```
uint8_t flag;
...
ISR(SOME_vect) {
  flag = 1;
}
...

        while (flag == 0) {
                ...
        }
```

the compiler will typically access `flag` only once, and optimize further accesses completely away, since its code path analysis shows that nothing inside the loop could change the value of `flag` anyway. To tell the compiler that this variable could be changed outside the scope of its code path analysis (e. g. from within an interrupt routine), the variable needs to be declared like:

```
volatile uint8_t flag;
```

ΠΑΡΑΡΤΗΜΑ ΄Δ

Στο παράρτημα αυτό περιλαμβάνονται δυο εισαγωγικά αποσπάσματα πάνω στην σύνταξη της γλώσσας C και στην χρήση της στο περιβάλλον WINAVR.

Why C instead ASM

ASM is a specific language as it is a low level programming language. It is mnemonics to a mashine codes. It takes tons of time to develop embedded programs in ASM language. Now even 8 bit microcontrollers are more capable as they were earlier. The program memories are going up to megabyte(s). Programs becoming more complex, functionality grows up. This is one reason to use higher level programming languages like C.

If using C language you do not have to go into details how processor works. You don't have to think about hardware logic. It is better to leave this work to C compiler which may help you to avoid bugs in silicon level.

Another C language benefit against ASM language is portability. If you work on one embedded system architecture and decide to move to other maybe more advanced. But if your previous program were written in ASM language, then you will have to rewrite this code from beginning. Using C language you can compile program for different microcontrollers without significant modifications of code. This way you new project code upgrade becomes ease.

When using C or other high level language it is easy to save specific hardware routines to libraries which are convenient to reuse in other projects. Using C libraries ensure that your application can be recompiled for different MCU's.

The most important benefit of using high level programming language like C is that you can focus on algorithm design and spend less time on implementing. C is a high level language. It enables to write, understand and maintain embedded programs faster and easier as one C language line can stand for several ASM lines.

ASM language can also be used in critical parts of algorithms, but again C compilers are improving and sometimes program written in C may be more efficient then written in ASM. It depends more on developer than a programming language.

The very basics of C

C language is function based programming language. C program itself is a function. Usually parameters to C function are passed as arguments. Function consists of a name followed by the parentheses enclosing arguments or an empty pair of parentheses if there are not arguments required. If there are several arguments, they are separated by commas.

The mandatory part in C program is *main* function. This function must be included in every program because this is a first function which is run after execution of program.

Lets take an example:

```
/*****************
```

```
#include <stdio.h>

int main(void)

{

printf("Hello world!\n");

return 0;

}

/*****************
```

This is a very basic C program, but it contains all necessary elements of it. Lets examine a little bit what we have written here...

#include <stdio.h> - is a preprocessor command. All preprocessor commands are identified by # sign at the beginning of the line. *#include* command tells preprocessor to open file *stdio.h* and from it necessary stored parts while compiling program. The file name is surrounded by brackets "<>". Bracket marks"<>" tels the preprocessor to search for the file in the region defined by operating system variables (for instance "path" variabel in Environment variables). If double quotes are used instead of bracket marks then file searching is done only in default directory of project.

Next line is *int main(void).* It is always necessary to define the type of function return type and of course function has an arguments. The type *int* is showing that *main* function returns an integer and no arguments to the function is needed.

Opening brace "{" indicates the beginning of the block of sentences. On of those sentences is printf(). This function is taken from stdio.h library and writes a message to computer terminal (this case screen).

Return 0 is returning parameter to the function. And every function should end with closing brace "}".

"/*" is commenting the line. This means that line marked with "/*"is not included in compilation.

Variables in embedded C language

What are variables in C language. Variables are simple keywords which are defined by the values. Values can be changed. Variables are like a boxes with some size where values like apples can be put in. So variables can be various forms and sizes so called variable types.

Variable type is defined by a reserved word which indicates the type and size of variable identifier:

unsigned char my_char;

long int all_my_numbers;

int number;

Why do we need variables? The basic answer is that memory is limited and compiler needs to know much space to reserve for each variable. So the programmer needs to specify the variable type and its size by using one of reserved words from the table:

Type	Description	Size	Domain
char	Signed character/byte. Characters are enclosed in **single** quotes.	1	-128..127
double	Double precision number	8	ca. $10^{-308}..10^{308}$
int	Signed integer	4	$-2^{31}..2^{31}-1$
float	Floating point number	4	ca. $10^{-38}..10^{38}$
long (int)	Signed long integer	4	$-2^{31}..2^{31}-1$
long long (int)	Signed very long integer	8	$-2^{63}..2^{63}-1$
short (int)	Short integer	2	$-2^{15}..2^{15}-1$
unsigned char	Unsigned character/byte	1	0..255
unsigned (int)	Unsigned integer	4	$0..2^{32}-1$
unsigned long (int)	Unsigned long integer	4	$0..2^{32}-1$
unsigned long long (int)	Unsigned very long integer	8	$0..2^{64}-1$
unsigned short (int)	Unsigned short integer	2	$0..2^{16}-1$

Few statements demonstrating the usage of variables and value assignments:

```
{
   int i;       /* Define a global variable i */
   i = 1;       /* Assign i the value 0 */
   {            /* Begin new block of sentences */
     int i;     /* Define a local variable i */
     i = 2;     /* Set its value to 2 */
   }            /* Close block */
   /* Here i is again 1 from the outer block */
}
```

Constants in C language

Constant value is understandable as non changeable value like PI=3.141592... value in math. Usually you use constants in your programs, but don't realize that they are constants. For instance:

x=x+3; The number 3 is a constant which will be compiled directly in addition operation. Constants can be character or string. Like in function printf("Hello World\n"); "Hello World" is a string constant which is placed in program memory and will never changes.

It is usually recommended to declare constants by using identifier with reserved word const:

const int No=44;

By identifying the variable as constant will cause compiler to store this variable in program memory rather than in RAM, thus saving space in RAM. If special functions used, then constants can be also stored in EEPROM memory.

Few words for the numeric constants. Numeric constants can be declared in many ways indicating their base.

- **Decimal integer constants** (base 10) consist of one or more digits, 0 through 9, where 0 cannot be used as the first digit.
- **Binary constants** (base 2) begin with a 0b or 0B prefix, followed by one or more binary digits (0, 1).
- **Octal constants** (base 8) consist of one or more digits 0 through 7, where the first digit is 0.
- **Hexadecimal constants** (base 16) begin with a 0x or 0X prefix, followed by a hexadecimal number represented by a combination of digits 0 through 9, and characters A through F.
- **Floating-point constants** consist of:
    - an optional sign - or +;
    - an integer part a combination of digits 0 through 9;
    - a period;
    - a fractional part a sequence of digits 0 through 9;
    - an optional exponent e or E, followed by an optionally signed sequence of one or more digits

For example, the following floating-point constant contains both the optional and required parts: `+1.15e-12`.

Character constant can be represented as character in quotation marks like 't' or as character code like '\x74'. Backslash may confuse compiler. Lets say you want to assign backslash as character then you should write like this '\\'.

C language Operators and expressions

The main thing what microcontrollers does is operates with data. There are four main operations that microcontrollers does: adds, abstracts, multiplies and divides (+,-,*,/). division can be split in division "/" and modulus operation "%". For instance i1/i2 is integer division.

Other part of operators is Relation operators. They are used for boolean conditions and expressions. Expressions with these operators return true or false values. Zero is taken as false and non zero value as true. Operators may be as follows: <, <=, > >=, ==, !=.

The priority of the first four operators is higher than that of the later two operators. These operators are used in relational expressions such as:

```
7 > 12        // false
20.1 < 20.2      // true
'b' < 'c'    // true
"abb" < "abc"  // true
```

Note that the equality operator is == and not =. '=' is an assignment operator.

If you want to compare a and b for equality then you should write a == b, not a = b because a = b means you are assigning the value of b to a, as shown in.

Next part of operators are logical operators. With logical operators results may be combined to get some logical results. Logical operators may be: negation "!", logical AND "&&" and logical OR "||".

R1	R2	R1 && R2	R1 || R2	! R1
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Next would be ternary operators return values based on the outcomes of relational expressions. For example, if you want to return the value of 1 if the expression is true and 2 if it is false, you can use the ternary operator.

Example

If you want to assign the maximum values of i and j to k then you can write the statement

```
k = ( i>j ) ? i : j;
```

If i > j then k will get the value equal to i, otherwise it will get the value equal to j.

The general form of the ternary operator is:

```
(expr 1) ? expr2 : expr3
```

If expr1 returns true then the value of expr2 is returned as a result; otherwise the value of expr3 is returned.

Incremental operators are used to increment or decrement value of variable. Incremental operators can be applied only to variables as prefix or postfix:

i = 3;

```
j =4;
k = i++ + --j;
```

you will get the value of k as 6, i as 4 and j as 3. The order of evaluation is as follows:

1. i gets the value 3.
2. j is decremented to 3.
3. k gets the value 3 + 3.
4. i is incremented

Another important operator when programming microcontrollers is bitwise operators. Bitwise operators interprets operands as strings of bits. Bitwise operators are six: bitwise AND(&), bitwise OR(|), bitwise XOR(^), bitwise complement(~), left shift(<<), and right shift(>>). For instance:

x >> 3 causes the variable x to be shifted to the right by three bits prior to its use.

r = r | 0x0c; The hexadecimal number 0x0c is a turned on and all other bits turned off.

Operator precedence rules

Operators	**Order of evaluation**	**Remarks**
[] ( ) ->	Left to right	Array subscript, function call
– + sizeof( ) ! ++ ––		
& * ~ (cast)	Right to left	Unary
* / %	Left to right	Binary Multiplicative
+ -	Left to right	Binary Additive
>> <<	Left to right	Shift operators

Operators	Order of evaluation	Remarks
< <= > >=	Left to right	Relational operators
== !=	Left to right	Equality operators
&	Left to right	Bitwise AND operator
^	Left to right	Bitwise XOR operator
|	Left to right	Bitwise OR operator
&&	Left to right	Logical And operator
||	Left to right	Logical Or operator
?:	Left to right	Conditional operator
= += -= *= /= %=		
&= -= |= <<= >>=	Right to left	Assignment
,	Right to left	Comma

Program Flow In Embedded C

Program Flow and control is a control method of your program. For example Loop constructions control repeated execution of repeated program segments where control is taken by control parameter. In this article we will go through if/else switch/case statements and loop sentences like while, do while, for.

While statement

As I mentioned three looping sentences available in C language one of them is While sentence. Lets take an example:

```
#include <stdio.h>

int main(void)

{

int guess, i;

i=1;

guess=5;

while (guess!=i)

{
```

```
i=guess;

guess=(i+(10000/i))/2;

}

printf("Square root of 10000 is %d\n", guess);

return 0;

}
```

While(guess!=i) invokes looping operation. This causes statement to be executed repeatedly at the beginning the condition guess!=i is checked. As long the argument is TRUE the while sentence will be continued continuously. When guess becomes equal to i while statement will be skipped. I am not going to deep in to it as there are tons of information about basic C.

For Loop

As we didn't know how many times the loop should be executed then we used while sentence, but when we know exactly hof many times we need to execute a sentence, there is another loop sentence for(s1;s2;s3). For construct takes 3 arguments each separated by semicolons. Without any theories lets see an example:

```
for (i = 0; i < 5; i++)

{

printf("value of i");

}
```

1. The for loop has four components; three are given in parentheses and one in the loop body.
2. All three components between the parentheses are optional.
3. The initialization part is executed first and only once.
4. The condition is evaluated before the loop body is executed. If the condition is false then the loop body is not executed.
5. The update part is executed only after the loop body is executed and is generally used for updating the loop variables.
6. The absence of a condition is taken as true.
7. It is the responsibility of the programmer to make sure the condition is false after certain iterations.

Do While sentence

do while is somehow similar to while just one difference that there testing of conditions is done after execution of first loop:

```
do

{

printf(" the value of i is %d\n", i);

i = i + 1;

}

while (i<5)
```

so

1. The loop body is executed at least once.
2. The condition is checked after executing the loop body once.
3. If the condition is false then the loop is terminated.
4. In this example, the last value of i is printed as 5.

If Else Statement

```
if(expression)

statement1

else

statement2
```

Lets consider a fragment of program:

```
while((c=getchar())!=EOF)

if(c>'0'&&c<'9')

n++;

else

n--;
```

I bet there is no need of explanation. Simply if first part true then first sentence is executed else other sentence is executed.

Other case is with if else if sentence structure:

if (condition 1)

simple or compound statement // s1

else if (condition 2)

simple or compound statement // s2

else if (condition 3)

simple or compound statement // s3

.....

else if (conditon n)

simple or compound statement // sn

Remember that:

1. You can use if-else if when you want to check several conditions but still execute one statement.
2. When writing an if-else if statement, be careful to associate your else statement to the appropriate if statement.
3. You must have parentheses around the condition.
4. You must have a semicolon or right brace before the else statement.

Switch statement

switch (expressions)

{

case constant expressions

}

For example:

switch (i/10)

{

case 0: printf ("Number less than 10"); // A

break;

case 1: printf ("Number less than 20"); // B

break;

case 2: printf ("Number less than 30"); // C

break;

default: printf ("Number greater than or equal to 40"); // D

break; }

Remember that:

1. The switch expression should be an integer expression and, when evaluated, it must have an integer value.
2. The case constant expression must represent a particular integer value and no two case expressions should have the same value.
3. The value of the switch expression is compared with the case constant expression in the order specified, that is, from the top down.
4. The execution begins from the case where the switch expression is matched and it flows downward.
5. In the absence of a break statement, all statements that are followed by matched cases are executed. So, if you don't include a break statement and the number is 5, then all the statements A, B, C, and D are executed.
6. If there is no matched case then the default is executed. You can have either zero or one default statement.
7. In the case of a nested switch statement, the break statements break the inner switch statement.
8. The switch statement is preferable to multiple if statements.

AVR GCC Structures

Basically Structures are nothing more than collection of variables so called members. Structures allows to reference all members by single name. Variables within a structure doesn't have to be the same type. General structure declaration:

```
struct structure_tag_name{
        type member1;
        type member2;
        ...
        type memberX
};
```

or

```
struct structure_tag_name{
        type member1;
        type member2;
        ...
        type memberX
} structure_variable_name;
```

in second example we declared the variable name. Otherwise variables can be declared this way:

struct structure_tag_name var1, var2,var3[3];

Members of structure can be accessed by using member operator (.). The member operator connects the member name to the structure.

Lets take an example:

```
struct position{
        int x;
        int y;
}robot;
```

we can set robot position by using following sentence:

```
robot.x=10;
robot.y=15;
```

or simply

```
robot={10,15};
```

Structures can be nested:

```
struct status{
        int power;
                struct position coordinates;
} robotstatus;
```

to access robot x coordinate we have to write:

```
x=robotstatus.coordinates.x;
```

Actions can be taken with structures:

- Copy;
- Assign;
- Take its address with &;
- Access members.

Of course you can treat a structure like a variable type. So you can create an array of structures like:

```
struct status{
        int power;
        struct position coordinates;
} robotstatus[100];
```

Accessing 15th robot power would be like this:

```
pow=robotstatus[15].power;
```

Pointers to structures in embedded C

Sometimes you might like to manipulate the members of structures in more generalized way. This can be done by using pointer reference to structure. Sometimes it is easier to pass pointer to structure than passing entire structure to function.

Structure pointer can be declared very easily:

```
struct structure_tag_name *variable;
```

For instance:

```
struct position{
        int x;
        int y;
}pos1, *pos2;
```

we can assign first structure address to structure pointer like this:

```
pos2=&pos1;
```

now we can access first structure members by using structure pointer:

```
pos2->x=10;
```

or we can write

```
(*pos2).x=10;
```

Structures can contain member pointers to other structures or to structures of same tyme. This way we can create lists – dynamic structure:

```
struct list{
        int listNo;
        struct list *next;      //pointer to another structure
}item1, item2;
```

This way you can assign the pointer with address to another structure:

```
item1.next=&item2;
```

Read more about dynamic structures in C tutorials.

Enumeration of variables

This is similar to #define preprocessor where you can describe a set of constatnts. Using preprocessor we use:

#define zero 0

#define one 1

#define two 3

But there is an alternative of using enumerating using keyword *enum*:

enum (zero=0,one, two); //zero=0, one=1; two=2

By default enumeration assigns values from 0 and up.

Now you can use enumeration like in following example:

int n;

enum (zero=0,one, two);

n=one; //n=1

You can use enum like this:

enum escapes { BELL = '\a', BACKSPACE = '\b', HTAB = '\',

RETURN = '\r', NEWLINE = '\n', VTAB = '\v' };

or

enum boolean { FALSE = 0, TRUE };

An advantage of **enum** over **#define** is that it has scope This means that the variable (just like any other) is only visible within the block it was declared within.

Example:

```
main()
{
enum months {Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,
Dec};
/*     A      A
       |      |
       |      |
       |      -------  list of aliases.
       --------------  Enumeration tag.      */

        enum months month;     /* define 'month' variable of type
'months' */

        printf("%d\n", month=Sep);    /* Assign integer value via an
alias
                                  * This will return a 9 */
}
```

Typecasting in AVR-GCC

I am sure- typecasting is one of common practices in embedded C when converting one variable to another. As AVR microcontrollers are 8 bit systems and sometimes we operate with 16, 32 or even 64 bit length variables we have an ability to change like 16 bit integer value to 8 bit char and so on. This operations is so called typecasting. So it is important to understand this operation in order to use it properly and get desired result.

When you want to typecast a variable use parenthesis symbols "(" and ")" like in following example:

int X;

char C=30;

*X=(int)C*10; //variable C is type-casted to int*

Now X result is 300. Without typecasting compiler would think C as char and result would be 44(0x2C) as maximum value for char is 255.

Do not ignore C compiler pre-processor

You maybe don't know or probably didn't think about this but you are using program preprocessing before compiling it. As I said before compiling I mean, that each time you are compiling your project, C compiler prepares program file to be ready for compilation. Preprocessor includes other files to main file, defines symbol constants and macros, prepares conditional compilation of code. All preprocessor tasks are marked with ampersand symbol "#". Lets go through most of directives of preprocessor:

#include

One and common directive is #include. You have already noticed, that you use this directive to include external c files and headers to your project. You can use #include in two ways:

#include <filename>
#include "filename"

The difference between them is where preprocessor will be looking for these files. If "< >" are used then preprocessor will be looking for the standard library if "" "" then preprocessor will look in default project folder or location defined in the makefile.

#define

#define is a second important directive which allows creation of constants and macro commands. Structure of usage define is:

#define identifier value

When constant or macro command is defined, then in any place where Identifier is used it will be changed to defined value. For instance

#define PI 3.14159

Then in program PI value will be replaced by 3.14159. This allows to control values very easily. It could be some particular constants, port address definitions. And of course usage of good nomenclature of identifiers makes program more readable and descriptive.
#define can also be used to define macro commands. Following example will clear things out:

#define CIRCLE_AREA(x) (PI(x)*(x))*

Each time when in program CIRCLE_AREA(x) will be used, identifier will be replaced with its value like:

area=CIRCLE_AREA(4);

will be replaced by:

*area=(3.14159*4*4);*

What is benefit of this? First of all when regular functions are used, then program calls them from specific address. Using macro commands functions are inserted in-line. Sometimes it may speed up program.
Opposite to #define directive is #undef directive which removes previous definition of constant or macro.

Conditional compilation
When you want to make your program more universal, then you can make perform conditional compilation. For instance if you want your program work with with multiple AVR mocrocontrollers then you cannot avoid conditional compilation depending on which processor is used. Simple example helps you understand the matter:

```
#if !defined(null)
#define NULL 0
#endif
```

This is explained as follows. If NULL isn't define then define it. It depends on particular compiler but usually there are short variants of such commands. For instance: #ifdef – stands for (#if defined) and #ifndef – stands for (#if !defined)or

#elif – (else if). Conditional definitions may be powerful tool in debugging process. For instance you may define:

#define DEBUG 1

Then in program you can add following lines in any place which can be turned on and off by changing DEBUG identifier value:

```
#ifdef DEBUG
printf(variable x=%d\n",x);
#endif
```

line printf() will be compiled if DEBUG identifier will have value 1 and line will be skipped if value will be 0.

#error

The command #error causes the preprocessor to report a fatal error. The rest of the line that follows #error is used as the error message.
You would use #error inside of a conditional that detects a combination of parameters which you know the program does not properly support.

```
#ifdef DEBUG
#error debug mode doesn't work here
#endif
```

#warning

The command #warning causes the preprocessor to report a warning but doesn't stop compilation like error command.
For more preprocessor commands and and usage refer to GCC documentation.

Accessing AVR microcontroller ports with WinAVR GCC

All AVR ports have Read-modify-write functionality when used as genera I/O ports. Direction of separate port pin can be changed. Each pin buffer has symmetric capability to drive and sink source. Pin driver is strong enough to drive LED directly , but it is not recommended. All port pins have selectable pull-up resistors. All pins have protection diodes to both VCC and GND.

Each port consists of three registers DDRx, PORTx and PINx (where x means port letter). DDRx register selects direction of port pins. If logic one is written to DDRx then port is configured to be as output. Zero means that port is configured as input. If DDRx is written zero and PORTx is written logic "1" then port is configured as input with internal pull-up resistor. Otherwise if PORTx is written to zero, then port is configured as input but pins are set to tri-state and you might need to connect external pull-up resistors.

If PORTx is written to logic "1" and DDRx is set to "1", then port pin is driven high. And if PORTx=0, then port is driven low.

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Lets move to C. How we can control AVR port pins? When using WinAVR GCC first we need to set up proper library where PORT register names have their addresses defined. Main library where general purpose registers are defined is io.h located in avr directory of WINAVR installation location.

```
#include <avr/io.h>
```

Now we can use port names instead of their addresses. For instance if we want to set all pins of PORTD as output we simply write:

```
DDRD=0xFF; //set port D pins as outputs
```

Now we can output a number to port D:

PORTD=0x0F; //port pins will be set to 00001111

if we have 8 bit variable i we can assign this variable to port register like this:

uint8_t i=0x54;

PORTD=i;

Lets read port D pins:

DDRD=0; //set all port D pins as input

i=PIND; //read all 8 pin bits of port Dand store to variable i

There is ability to access separate port pins. So all eight port pins can be used for multiple purposes.

Some of the pins may be configured as outputs and some as inputs and performs different functions.

Lets say we need 0,2,4,6 pins to be as input and 1,3,5,7 as output. Then we do like this:

DDRD=0; //reset all bits to zero

DDRD |=(1<<1)|(1<<3)|(1<<5)|(1<<7); //using bit shift "<<"operation and logical OR to set bits 1,3,5,7 to "1"

So we can output values to 1,3,5 and 7 pins

PORTD |=(1<<1)|(1<<3)|(1<<5)|(1<<7);

Or clear them

PORTD &=~((1<<1)|(1<<3)|(1<<5)|(1<<7));

Reading of port pins is easy. Set any pin(s) for input like this:

DDRD &=~((1<<1)|(1<<3)); //This clears bits 1 and 3 of port direction register

i=PIND; //reads all 8 pins of port D

You can read 1 and 3 bits by using masks or simply shift i value by 1 ar 3 positions to compare LSB with 1. Of course there are some functions in sfr_defs.h library like bit_is_set() or bit_is_clear() to check particular bits and make these tasks little easier.

Following example should clarify some issues:

```
#include "avr\io.h"
#include "avr\iom8.h"
int main(void) {
  DDRD&=~_BV(0);//set PORTD pin0 to zero as input
  PORTD|=_BV(0);//Enable pull up
  PORTD|=_BV(1);//led OFF
  DDRD|=_BV(1);//set PORTD pin1 to one as output
  while(1) {
   if (bit_is_clear(PIND, 0))//if button is pressed
                {
                        PORTD&=~_BV(1);//led ON
                        loop_until_bit_is_set(PIND, 0);//LED ON while
Button is pressd
                        PORTD|=_BV(1);//led OFF
                }
        }
}
```

Some notes about makefiles

If you aren't familiar with linux programming or compilation chances are this might be the first time you have encountered makefiles. Makefiles and a necesary component of every project. They are a form of batch file or script, they contain all a list of all the steps needed to take your code and turn it into a .hex file to be sent to the AT90s2313. In this way they are a form of program themselves.

To compile the average project with WINAVR you run a program called "make.exe". This proram by default looks for a file called "`makefile`" (note: no extension as it originated in the Unix environment) and reads the contents of that file to know which actions to perform to compile your source code. The "make.exe" program performs the steps listed in the file "`makefile`" to run other programs (compilers, linkers, etc). The file "`makefile`" will also store the necessary command-line options that need to be specified when invoking other programs (compilers, linkers, etc). Thus it is entirely possible to not use a `makefile` and go through the process of manually running each program that needs to act upon your source code to generate the final .hex file to program into the AT90s2313. The advantage of the `makefile` is that you can store these repetative processes in a list called the `makefile` and have a program called "`make.exe`" do all the steps you've listed in the `makefile`.

If the thought of spending the time working out each step required to compile your program and the options that need to be specified at each step doesn't appeal to you then you are in luck. Most of the time, at least for simple programs, you can copy a `makefile` from somewhere else, modify the few lines that need changing and compile your source code. The AVRFreaks distribution of WINAVR has several makefiles already included in it. The one I will be using is located in the "sample" directory, for example "`C:\WINAVR\sample\makefile`". I have also included a copy here (right-click and "save as" if you want to download it)

For each project you need a separate copy the `makefile` with it. So when starting a new project for the examples here, do not forget to copy the above `makefile` into the same directory as your source files (mysource.c and mysource.h for example). When you have you will need to edit just two lines to get simple programs to compile.

Firstly scroll down until you find these lines;

```
# MCU name

MCU = atmega128
```

and change them so that instead of the atmega128 being the target AVR the AT90s2313 is. For example

```
# MCU name

MCU = at90s2313
```

That was the first alteration. The other is just below this one where you see:

```
# Target file name (without extension).

TARGET = main
```

This needs to be changed to reflect the name you have chosen for your main source file. For example if you were writing a program to control an LCD and called the .c source file "lcd.c" then you need to replace "main" with "lcd". For example:

```
# Target file name (without extension).

TARGET = lcd
```

These two changes means you have directed the `makefile` to (1) compile for the AT90s2313 and (2) use your .c file to create the code. Now if you invoke "`make.exe`" program in the directory of you source code and your new `makefile` you (hopefully) will get no errors and have a .hex file ready for programming into the physical AT90s2313. If it does not work for you, double check you have set up the "path" variable correctly in windows so that it includes the make.exe file and that also the directory where you installed WinAVR is also on the "path" variable. These topics are covered in the installation guide that comes with the AVRFreaks WinAVR distribution (found here).

I also highly recommend reading through the beginning of the official 'GNU make' manual, http://www.gnu.org/software/make/manual/html_mono/make.html.gz

Making a pin single go low

Basically the command is:

PORTD &= ~(1 << 5);

This line makes the sixth pin on PORTD go low. Why the sixth (6)? Because the first pin is 0. Therefore the second pin is called 1, the third pin is called 2 and therefore pin 6 is called 5. If this is confusing, which it can be at first, run the sample program and simulate it in AVR Studio and then change the value and see which pin is taken low. If you wanted to make the first pin of PORTD go low you would use

PORTD &= ~(1<<0);

How does it work? Well to the beginner this can look quite daunting but it is all quite simple.

Note this is the prefered method now as old commands supported by C such as cbi() amd sbi() have now be deprecated (dropped). I mention this here as some old code around may use cbi() or sbi(). These commands have actually been dropped from AVRGCC in the latest version, so any code using cbi() and sbi() will not compile. If you desparately want them still you'll need to write your own macro to re-implement them using the code presented here as the basic.

Anyway, onto the explanation.

&= operator

Firstly, the "&=" part.

PORTD &= ~(1 << 5); is excactly the same as *PORTD = PORTD & ~(1<<5);*
The "&=" is a shorthand way of writing it. It just means PORTD equals PORTD AND something.

The "&" part is the logical bitwise AND operator. This takes two binary numbers (1 or 0) and returns 1 if both are 1 or 0 for everything else.

e.g.

1 & 1 = 1

1 & 0 = 0

0 & 1 = 0

0 & 0 = 0

When operating on longer numbers it works on each pair of bits individually.

e.g.

11001100
11111111 &
11001100

11001100
01010101 &
01000100

~ (tilde) operator

Now the ~ (tilde). This is the "One's Complement" operator. Also known as logic NOT. Basically it takes 0 and returns 1, or takes 1 and returns 0. It just creates the opposite of whatever is to the right of it.

<< (Shift Left) operator

Finally, the << operator. This is called the "Shift left" operator. This operator exists in all ANSI compliant C compilers but is not usually highly utilised with PCs etc. It is great for microcontrollers though. The "Shift left" operator is interesting in that it takes a binary (1 and 0) number and shifts everything left and brings in 0s from the right.

The form of it is:

value << number of bit positions

e.g.

0101 << 1 would result in *1010*

00100000 << 2 would result in *10000000*

00000001 << 4 would reuslt in *00010000*

Putting it all together (proof)

If PORTD was 11111111 before this command it would be 11011111 afterwards. **Remember that pin 0 is the right most binary digit**.

How we get that is as follows:

PORTD starts as 11111111 (0xFF is the hex value for this)

(1<<5) is the same as *(00000001 << 5)* and works out to be *00100000*

So we could now write:

PORTD &= ~(00100000)

Now I'll expand the equation to get rid of the "&=" part. (See above how it works).

Getting rid of "&=" gives us:

PORTD = PORTD & ~(00100000)

Now well will get rid of the ~ (tilde):

PORTD = PORTD & 11011111

Now we step away from proper mathematics and into programming order of operations.

PORTD starts out as 11111111 so we can substitute that for the second PORTD in our equation. (This is only because this is how the microprocessor and most computers do it). So this gives us:

PORTD = 11111111 & 11011111

Performing the & (AND) operator gives us:

PORTD = 11011111

And this is what the port will set at.

NOTE: By doing all these steps, only the one individual bit is changed and all others are left as they were

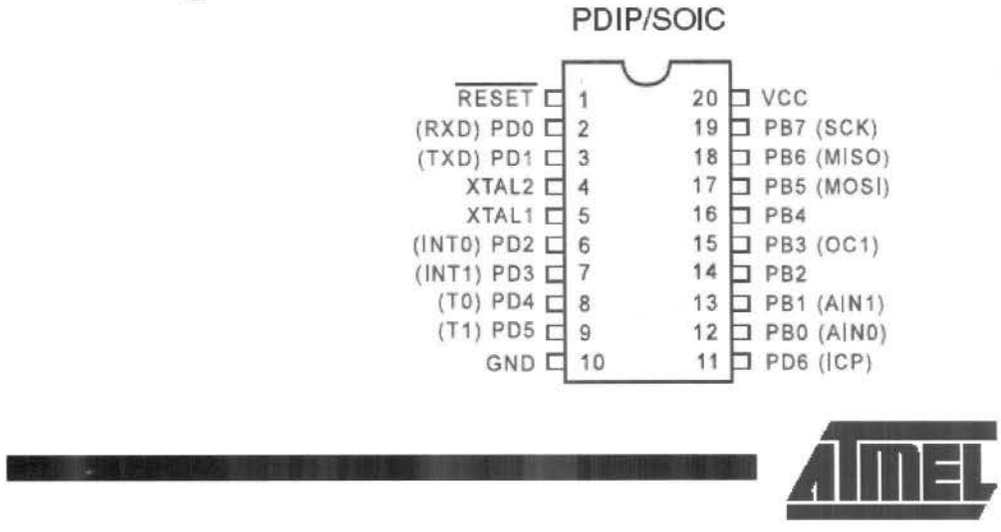
Therefore we made the sixth and only the sixth bit go low. I suggest you play around with the example code and see how changing it affects the output in the simualtion in AVR studio.

Making a pin go high

I found this great description of AVR I/O in the Proycon AVRLib. Click here to view a text file with the extract explaining AVR I/O.

Once you've read that description look here at the pin configuration of the AT90s2313. This is taken from the datasheet

Pin Configuration



You can see Port B running from pin number 12 to pin 19. In our example program we will make pin PB2 (pin 14) go high (5 volts).

By including *#include <avr/io.h>* in our code, we can access Port B by changing the values stored at PORTB.

Now, the needed line of code is:

PORTB |=(1<<2);

'|' Bitwise OR operator

The | operater is the bitwise 'OR' operator. Which looks at a pair of 1s or 0s and returns true (1) if one or both of the test set contain a 1.

10010001
11001100 'bitwise OR'

11011101

'|=' operator

In C, there are lots of shortcuts. One of them is adding an 'equals' sign after an operator. Any operator.

The effect of doing this is to shorten the following statement:

PORTB = PORTB | something

To just:

PORTB |= something

Both statements are the same and do the same thing.

'<<' Shift-Left operator

The << operator. This is called the "Shift left" operator. This operator exists in all ANSI compliant C compilers but is not usually highly utilised with PCs etc. It is great for microcontrollers though. The "Shift left" operator is interesting in that it takes a binary (1 and 0) number and shifts everything left and brings in 0s from the right.

The form of it is:

value << number of bit positions

e.g.

0101 << 1 would result in 1010

00100000 << 2 would result in 10000000

00000001 << 4 would reuslt in 00010000

Thus in the example line of code we have for making a pin go high, the expression (1<<2) is the same as (00000001 << 2) and results in 00000100. The reason we use it is so when we want it to change another pin we just change the 2 to whatever we want. This allows us to not hardcode values if we want to not hardcode values. i.e. the '2' is pulled from a file of another function etc.

Putting it all together (Proof)

If PORTB was 00110011 to begin with and then we ran the line of code:

PORTB |=(1<<2);

The following would happen.

(1<<2) would be evaluated to 00000100 so we could instead write:

PORTB |=(00000100);

Another step would be to expand the |= shortcut, therefore:

PORTB = PORTB | (00000100);

Now the 'PORTB' on the right hand side of the equation represents what the value is, the one on the left is what we want it to be (the result will be stored in the left hand side, thus the left PORTB represents the final result of the statement and changes PORTB to reflect this).

Thus we can substitute the 'before' value of PORTB in for the right hand side 'PORTB':

PORTB = 00110011 | (00000100);

This becomes:

PORTB = 00110111;

The underlines bit is the one that has changed.

Therefore the single bit that represents the third (PB2) pin on PORTB has been made to go high and no other pins have changed.

Any errors, questions, comments etc can be posted in the forum or sent via the emailpage.

```
// All AVR processors have I/O ports which each contain up to 8
        // user-controllable pins.  From a hardware perspective,
these I/O pins
        // are each an actual physical pin coming out of the
processor chip.
        // The voltage on the pins can be sensed or controlled via
software,
        // hence their designation as Input/Output pins.

        // While I/O pins are actual wires in the real world, they
also exist
        // inside the processor as special memory locations called
registers.
        // The software-controlled contents of these registers is
what
        // determines the state and operation of I/O pins and I/O
ports.

        // Since AVR processors deal naturally with 8 bits at a time,
I/O pins
        // are grouped into sets of 8 to form I/O ports.  Three
registers
        // are assigned to each I/O port to control the function and
state of
        // that port's pins.  The registers are 8-bits wide, and each
bit (#0-7)
        // determines the operation of the corresponding number pin
(pin 0-7).
        // The three registers are:

        // DDRx  - this register determines the direction
(input/output) of the pins on port[x]
        //                      A '0' bit in the DDR makes that port
pin act as input
        //                      A '1' bit in the DDR makes that port
pin act as output

        // PORTx - this register contains the output state of the
pins on port[x]
        //                      A '0' bit makes the port pin output a
LOW  (~0V)
        //                      A '1' bit makes the port pin output a
HIGH (~5V)

        // PINx  - this register contains the input state of the pins
on port[x]
        //                      A '0' bit indicates that the port pin
is LOW  (at ~0V)
        //                      A '1' bit indicates that the port pin
is HIGH (at ~5V)

        // The x should be replaced with A,B,C,D,E,F, or G depending
on the
        // desired port.  Note that not all AVR processors have the
same set
        // or number of ports.  Consult the datasheet for your
specific processor
        // to find out which ports it has.
```

```
        // in the AVR-GCC C language, ports can be accessed using two
kinds of
        // commands:
        // inb() and outb()             -       in-byte and out-byte
        // cbi() and sbi()              -       clear-bit and set-bit

        // inb() and outb() should be used when you intend to read or
write
        // several bits of a register at once.  Here are some
examples:

        outb(DDRA, 0x00);       // set all port A pins to input
        a = inb(PINA);          // read the input state of all pins on
port A

        outb(DDRB, 0xFF);       // set all port B pins to output
        outb(PORTB, 0xF0);      // set PB4-7 to HIGH and PB0-3 to LOW

        // Often you may wish to change only a single bit in the
registers
        // while leaving the rest unaltered.  For this, use cbi() and
sbi().
        // For example:

        sbi(DDRC, 0);           // sets PC0 to be an output
        sbi(DDRC, 1);           // sets PC1 to be an output

        cbi(PORTC, 1);          // sets PC1 to output a LOW without
altering any other pin

        // the lines below will cause PC0 to pulse twice,
        // but will leave all other port C pins unchanged
        cbi(PORTC, 0);          // sets PC0 to output a LOW
        sbi(PORTC, 0);          // sets PC0 to output a HIGH
        cbi(PORTC, 0);          // sets PC0 to output a LOW
        sbi(PORTC, 0);          // sets PC0 to output a HIGH
        cbi(PORTC, 0);          // sets PC0 to output a LOW


        return 0;
}
```

ΠΑΡΑΡΤΗΜΑ Έ

Στο παράρτημα αυτό παραθέτω δυο αποσπάσματα από το βιβλίο του κ. Μρατάκου Φυσική Ι. Τα οποία αφορούν στην αεροδυναμική και στην περιστροφική κίνηση στερεού σώματος.

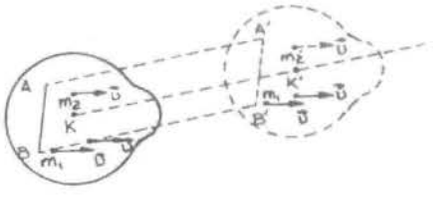
Κεφάλαιο 6ο

ΚΙΝΗΜΑΤΙΚΗ ΚΑΙ ΔΥΝΑΜΙΚΗ ΤΟΥ ΣΤΕΡΕΟΥ ΣΩΜΑΤΟΣ

6-1 Εἴδη κινήσεων στερεοῦ σώματος

Ἀπ' ὅλες τίς κινήσεις πού μπορεῖ νά κάνει ἕνα σῶμα οἱ ἁπλούστερες εἶναι ἡ μεταφορική, ἡ στροφική καί ἡ σύνθετη.

α) Μεταφορική κίνηση στερεοῦ σώματος. Ὀνομάζεται μ ε τ α φ ο ρι-κ ή κίνηση ἑνός στερεοῦ σώματος ἡ κίνηση, στήν ὁποία ὅλα τά σημεῖα τοῦ στερεοῦ σώματος ἔχουν, σέ κάθε χρονική στιγμή, τήν ἴδια ταχύτητα $\vec{\upsilon}$. Στήν κίνηση αὐτή, εἶναι δυνατό, βέβαια, οἱ ταχύτητες τῶν σημείων τοῦ σώματος νά ἀλλάζουν μέ τήν πάροδο τοῦ χρόνου· εἶναι, ὅμως, ἴσες μεταξύ τους τήν ἴδια χρονική στιγμή. Ἀπό τόν ὁρισμό τῆς μεταφορικῆς κινήσεως σώματος συμπεραίνεται ὅτι, κατά τήν κίνηση αὐτή, I) οἱ τροχιές ὅλων τῶν σημείων τοῦ σώματος εἶναι μεταξύ τους παράλληλες (Σχ.6-1), II) οἱ ἐπιταχύνσεις ὅλων τῶν σημείων τοῦ σώματος εἶναι μεταξύ τους ἴσες σέ κάθε χρονική στιγμή καί III) κάθε εὐθεία τοῦ σώματος, π.χ. ἡ ΑΒ (Σχ. 6-1), παραμένει συνεχῶς παράλληλη πρός τόν ἑαυτό της.

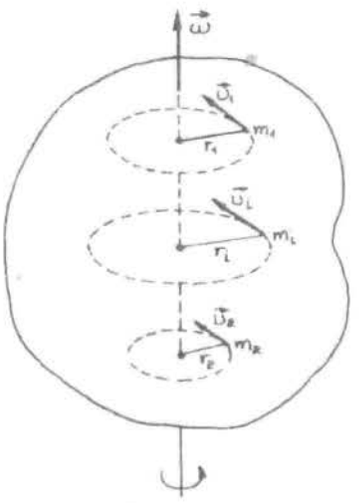


Σχ. 6-1. Κατά τή μεταφορική κίνηση, οἱ τροχιές ὅλων τῶν σημείων τοῦ σώματος εἶναι μεταξύ τους παράλληλες.

Σάν παράδειγμα μεταφορικῆς κινήσεως ἀναφέρουμε τήν κίνηση συρόμενου κιβωτίου καί τήν κίνηση μαγνητικῆς βελόνας στηριγμένης πάνω σέ κατακόρυφο ἄξονα, πού ἡ βάση του μετατοπίζεται πάνω σέ ὁριζόντιο ἐπίπεδο.

Ἀπό τά πιό πάνω προκύπτει ὅτι ἡ μελέτη τῆς μεταφορικῆς κινήσεως ἑνός σώματος ἀνάγεται στή σπουδή τῆς κινήσεως τοῦ κέντρου μάζας του, τό ὁποῖο θεωροῦμε σάν ὑλικό σημεῖο, πού ἔχει μάζα ἴση μέ τή μάζα τοῦ σώματος αὐτοῦ.

β) Στροφική κίνηση στερεοῦ σώματος. Ὀνομάζεται στροφική κίνηση ἑνός στερεοῦ σώματος ἡ κίνηση, στήν ὁποία ἕνα σύνολο σημείων τοῦ σώματος, πού βρίσκονται πάνω σέ μία εὐθεία, παραμένουν διαρκῶς ἀκίνητα (Σχ. 6-2). Ἡ εὐθεία αὐτή, πάνω στήν ὁποία βρίσκονται τά ἀκίνητα σημεῖα τοῦ στρεφόμενου σώματος, ὀνομάζεται ἄξονας περιστροφῆς του. Ὅλα τ' ἄλλα σημεῖα τοῦ σώματος ἐκτελοῦν κυκλικές κινήσεις, πού οἱ ταχύτητές τους εἶναι ἀνάλογες μέ τίς ἀποστάσεις τῶν σημείων αὐτῶν ἀπό τόν ἄξονα. Δηλαδή ἰσχύει

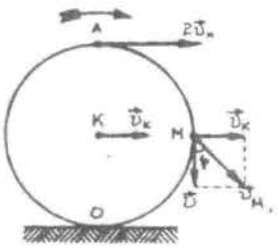


Σχ. 6-2.

$$\boxed{\upsilon = \omega . r,} \qquad (6\text{-}1)$$

ὅπου ω εἶναι ἡ γωνιακή ταχύτητα τοῦ σώματος, πού παραμένει ἡ ἴδια γιά ὅλα του τά σημεῖα σέ κάθε χρονική στιγμή.

γ) Σύνθετη κίνηση στερεοῦ σώματος. Ὀνομάζεται σύνθετη κίνηση ἑνός στερεοῦ σώματος, ἡ κίνηση, στήν ὁποία τό σῶμα ἐκτελεῖ σύγχρονα καί μεταφορική καί στροφική κίνηση. Γιά παράδειγμα τέτοιας κινήσεως ἔχομε τόν τροχό τοῦ αὐτοκινήτου, ὅταν αὐτός κυλιέται πάνω στό ἔδαφος, γιατί ἀπό τή μιά μέν αὐτός περιστρέφεται γύρω ἀπό τόν ἄξονά του ἀπό τήν ἄλλη δέ ἐκτελεῖ τήν μεταφορική κίνηση τοῦ αὐτοκινήτου.



Σχ. 6-3

Ἡ ταχύτητα $\vec{\upsilon}_M$, κάθε σημείου Μ τοῦ τροχοῦ, εἶναι ἴση μέ τό διανυσματικό ἄθροισμα τῆς ταχύτητας $\vec{\upsilon}_K$, πού ἔχει αὐτό λόγω τῆς μεταφορικῆς κινήσεως, καί τῆς ταχύτητας $\vec{\upsilon}$, πού ἔχει αὐτό λόγω τῆς στροφικῆς κινήσεως τοῦ τροχοῦ. Δηλαδή

$$\vec{\upsilon}_M = \vec{\upsilon}_K + \vec{\upsilon}. \qquad (6\text{-}2)$$

Ἀπ' αὐτή προκύπτει

$$\upsilon_M = \sqrt{\upsilon^2 + \upsilon_K^2 + 2\upsilon\upsilon_K \sigma\upsilon\nu 2\varphi}. \qquad (6\text{-}3)$$

Εἰδικά, ἡ συνισταμένη ταχύτητα τοῦ σημείου Ο εἶναι ἴση μέ μηδέν, γιατί, κατά τήν κύλιση, ὁ τροχός δέν ὀλισθαίνει στό ἔδαφος, ὁπότε ἔχουμε

$$\upsilon_K - \upsilon = 0. \qquad (6\text{-}4)$$

Οἱ Ἐξ. (6-3) καί (6-4) δίνουν,

$$\upsilon_M = \upsilon_K \sqrt{2(1+\sigma\upsilon\nu 2\varphi)}. \qquad (6\text{-}5)$$

Γιά τό ἀνώτατο σημεῖο Α τοῦ τροχοῦ ἡ συνισταμένη ταχύτητα θά εἶναι ἴση μέ

$$\upsilon_A = 2\upsilon_K. \qquad (6\text{-}6)$$

Εἶναι δυνατό νά ἀποδειχθεῖ ὅτι κάθε κίνηση ἑνός στερεοῦ σώματος, καί ἡ πιό πολύπλοκη, μπορεῖ νά θεωρηθεῖ σάν σύνθετη, πού προέρχεται ἀπό τή σύνθεση μιᾶς μεταφορικῆς καί μιᾶς στροφικῆς.

6-2 Κινητική ἐνέργεια στερεοῦ σώματος, πού κάνει μεταφορική κίνηση

Γιά τόν ὑπολογισμό τῆς κινητικῆς ἐνέργειας ἑνός στερεοῦ σώματος, πού κάνει μεταφορική κίνηση, θεωροῦμε ὅτι αὐτό ἀποτελεῖται ἀπό ἕνα πλῆθος ὑλικῶν σημείων, μέ μάζες $m_1, m_2, m_3, \ldots, m_i, \ldots$, καί ὑπολογίζουμε τήν κινητική ἐνέργεια καθενός ἀπό αὐτά· ὁπότε τό ἄθροισμα τῶν κινητικῶν ἐνεργειῶν ὅλων αὐτῶν τῶν ὑλικῶν σημείων θά μᾶς δώσει τήν ὁλική κινητική ἐνέργεια τοῦ σώματος. Δηλαδή ἰσχύει

$$T = \Sigma \frac{1}{2} m_i \upsilon_i^2, \qquad (6\text{-}7)$$

ὅπου $i=1,2,3,\ldots$ καί $\frac{1}{2} m_i \upsilon_i^2$ ἡ κινητική ἐνέργεια ὑλικοῦ σημείου τοῦ σώματος μάζας m_i. Ἀπό τήν Ἐξ. (6-7), ἐπειδή ὅλα τά σημεῖα τοῦ σώματος ἔχουν τήν ἴδια ταχύτητα υ, ἔχουμε

$$T = \frac{1}{2} \upsilon^2 \Sigma m_i. \qquad (6\text{-}8)$$

Ἐξάλλου ἐπειδή τό Σm_i ἐκφράζει τήν ὁλική μάζα m τοῦ σώματος, ἡ Ἐξ. (6-8) γίνεται

$$\boxed{T = \frac{1}{2} mv^2,} \qquad (6\text{-}9)$$

ἡ ὁποία δίνει τήν ὁλική κινητική ἐνέργεια στερεοῦ σώματος, πού κάνει μεταφορική κίνηση.

6-3 Κινητική ἐνέργεια στερεοῦ σώματος, πού κάνει στροφική κίνηση - Ροπή ἀδράνειας

Ἔστω στερεό σῶμα μάζας m, πού στρέφεται γύρω ἀπό σταθερό ἄξονα μέ σταθερή γωνιακή ταχύτητα ω (Σχ. 6-2). Ἡ κινητική ἐνέργεια κάθε σημείου τοῦ σώματος εἶναι ἴση μέ

$$T_i = \frac{1}{2} m_i v_i^2,$$

ὅπου m_i ἡ μάζα τοῦ ἀντίστοιχου σημείου i. Προφανῶς, ἡ ὁλική κινητική ἐνέργεια τοῦ σώματος θά εἶναι ἴση μέ

$$T = \Sigma T_i = \Sigma \frac{1}{2} m_i v_i^2. \qquad (6\text{-}10)$$

Ἐπειδή ἰσχύει

$$v_i = \omega_i r_i,$$

ἡ Ἐξ. (6-10) δίνει

$$T = \Sigma \frac{1}{2} m_i \omega_i^2 r_i^2 \qquad (6\text{-}11)$$

καί ἐπειδή στή στροφική κίνηση ὅλα τά σημεῖα τοῦ σώματος ἔχουν τήν ἴδια γωνιακή ταχύτητα ω, σέ κάθε χρονική στιγμή, ἡ Ἐξ. (6-11) γίνεται

$$T = \frac{1}{2} \omega^2 \Sigma m_i r_i^2 \qquad (6\text{-}12)$$

Στή σχέση (6-12) τό ἄθροισμα $\Sigma m_i r_i^2$ παίζει σπουδαῖο ρόλο στή μελέτη τῆς κινήσεως τοῦ στερεοῦ σώματος καί ὀνομάζεται **ροπή ἀδράνειας**

α ς I τοῦ σώματος ὡς πρός δοσμένο ἄξονα. Δηλαδή ἔχουμε

$$\boxed{I = \Sigma m_i r_i^2.} \qquad (6\text{-}13)$$

Ἀπό τίς Ἐξ. (6-12) καί (6-13) παίρνουμε

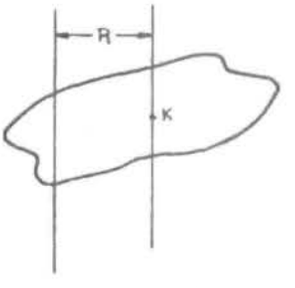
$$\boxed{T = \frac{1}{2} I\omega^2,} \qquad (6\text{-}14)$$

πού μᾶς δίνει τήν κινητική ἐνέργεια στρεφόμενου σώματος.

6-4 Θεώρημα Steiner - Μεταβολή τῆς ροπῆς ἀδράνειας

Ἀπό τήν Ἐξ. (6-13) βλέπουμε ὅτι ἡ ροπή ἀδράνειας ἑνός στερεοῦ σώματος ὡς πρός ἕναν ἄξονα περιστροφῆς ἐξαρτᾶται,ὄχι μόνο ἀπό τή μάζα του, ἀλλά καί ἀπό τό τρόπο κατανομῆς της γύρω ἀπό τό θεωρούμενο ἄξονα περιστροφῆς.

Σέ σχέση μέ τά πιό πάνω, ἀποδείχνεται θεωρητικά τό θ ε ώ ρ η μ α τ ο ῦ S t e i n e r, πού λέει ὅτι ἡ ροπή ἀδράνειας I ἑνός σώματος, ὡς πρός τυχόντα ἄξονα περιστροφῆς, εἶναι ἴση μέ τή ροπή ἀδράνειας $I_{κ,μ}$ τοῦ σώματος, ὡς πρός ἕναν ἄξονα περιστροφῆς πού περνᾶ ἀπό τό κέντρο μάζας του καί εἶναι παράλληλος πρός τόν τυχόντα,αὐξημένη κατά τό γινόμενο τῆς μάζας m τοῦ σώματος ἐπί τό τετράγωνο τῆς ἀποστάσεως R τῶν δύο τούτων ἀξόνων (Σχ. 6-4). Δηλαδή ἰσχύει



Σχ. 6-4.

$$\boxed{I = I_{κ,μ} + mR^2.} \qquad (6\text{-}15)$$

6-5 Ὑπολογισμός τῆς ροπῆς ἀδράνειας

Ἡ ροπή ἀδράνειας ἑνός στερεοῦ σώματος ὡς πρός δοσμένο ἄξονα ὑπολογίζεται γενικά μέ τή βοήθεια τοῦ ὁλοκληρώματος.

$$I = \int r^2 dm, \qquad (6\text{-}16)$$

στό ὁποῖο κάθε στοιχειώδης μάζα dm ἑνός στοιχείου τοῦ σώματος εἶναι πολλαπλασιασμένη ἐπί τό τετράγωνο τῆς ἀποστάσεως τοῦ στοιχείου ἀπό τόν ἄ-

ξονα. Η άθροιση στό πιό πάνω όλοκλήρωμα έπεκτείνεται σέ όλη τή μάζα τοῦ σώματος.

Από τή σχέση όρισμοῦ τῆς ροπῆς άδράνειας συμπεραίνουμε ὅτι αὐτή εἶναι μονόμετρο μέγεθος καί ἔχει τίς ἑξῆς μονάδες:

Στό CGS τό 1 $gr.cm^2$.

Στό MK τό 1 $kgr.m^2$.

Στό Τ.Σ τό 1 $kp.sec^2$.

Ο ὑπολογισμός τῆς ροπῆς ἀδράνειας ἑνός σώματος ἀπαιτεῖ τόν ὑπολογισμό τοῦ ὁλοκληρώματος τῆς Εξ. (6-16).

Ο πίνακας 6-1 δείχνει τίς ροπές ἀδράνειας ὁμογενῶν στερεῶν σωμάτων ὡς πρός ἄξονα, πού περνάει ἀπό τό κέντρο μάζας τοῦ σώματος.

Πίνακας 6-1

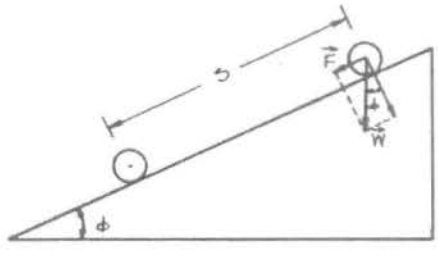
Σῶμα	Αξονας	Ροπή ἀδράνειας	
Ομοιόμορφη ράβδος μήκους l	Κάθετος στή ράβδο	$\frac{ml^2}{12}$	
Κυλινδρικός σωλήνας ἀκτίνας r	Αξονας κυλίνδρου	mr^2	
Συμπαγής κύλινδρος ἀκτίνας r	Αξονας κυλίνδρου	$\frac{mr^2}{2}$	
Συμπαγής σφαίρα ἀκτίνας r	Διάμετρος	$\frac{2}{5}mr^2$	
Κοίλη σφαίρα ἀκτίνας r	Διάμετρος	$\frac{2}{3}mr^2$	
Παχύτοιχος κυλινδρικός σωλήνας ἐσωτερικῆς ἀκτίνας r_1 καί ἐξωτερικῆς r_2	Αξονας κυλίνδρου	$\frac{m}{2}(r_1^2+r_2^2)$	
Ορθός κυκλικός κῶνος ὕψους h καί βάσεως ἀκτίνας r	Αξονας κώνου	$\frac{3}{10}mr^2$	
Στερεός κύλινδρος μήκους l καί ἀκτίνας r	Κεντρική διάμετρος	$\frac{1}{4}m(r^2+\frac{l^2}{3})$	

6-6 Κινητική ένέργεια στερεοῦ σώματος, πού κάνει σύνθετη κίνηση

Ἄν θεωρήσουμε ὅτι $υ_{κ,μ}$ εἶναι ἡ ταχύτητα τοῦ κέντρου μάζας ἑνός σώματος, λόγω μεταφορικῆς κινήσεώς του καί ὅτι ὁ ἄξονας στροφικῆς κινήσεώς του περνάει ἀπό τό κέντρο μάζας του. Ἡ ὁλική κινητική ἐνέργεια τοῦ σώματος θά εἶναι ἴση μέ τό ἄθροισμα τῆς κινητικῆς ἐνέργειας τοῦ σώματος, λόγω τῆς μεταφορικῆς κινήσεώς του καί τῆς κινητικῆς του ἐνέργειας, λόγω τῆς στροφικῆς κινήσεως αὐτοῦ γύρω ἀπό τό θεωρούμενο ἄξονα περιστροφῆς. Δηλαδή ἰσχύει

$$\boxed{T_{ολ} = \frac{1}{2} mυ^2_{κ,μ} + \frac{1}{2} I_{κ,μ}ω^2.} \qquad (6\text{-}17)$$

Παράδειγμα 6-1. Μιά σφαίρα κυλίεται πρός τά κάτω, πάνω σέ κεκλιμένο ἐπίπεδο γωνίας κλίσεως φ.Νά βρεθεῖ ἡ ταχύτητα τῆς σφαίρας, ὅταν διανύσει διάστημα s. Δίνεται ὅτι ἡ ροπή ἀδράνειας τῆς σφαίρας, ὡς πρός ἄξονα πού περνάει ἀπό τό κ.μ. της,εἶναι $I = \frac{2}{5} mr^2$ καί ὅτι οἱ τριβές θεωροῦνται ἀσήμαντες.



Σχ. 6-5

Λύση. Τό κ.μ. τῆς σφαίρας κατεβαίνει κάτω ἀπό τήν ἐπίδραση τῆς δυνάμεως

$$F = mgημφ,$$

πού στό διάστημα s δίνει ἔργο

$$W = mgsημφ. \qquad (6\text{-}18)$$

Τό ἔργο αὐτό μετατρέπεται σέ κινητική ἐνέργεια $\frac{1}{2} mυ^2$, λόγω τῆς μεταφορικῆς κινήσεως, καί $\frac{1}{2} Iω^2$, λόγω τῆς περιστροφικῆς κινήσεως τῆς σφαίρας γύρω ἀπό ἄξονα πού περνάει ἀπό τό κ.μ. της. Δηλαδή ἔχουμε

$$W = \frac{1}{2} mυ^2 + \frac{1}{2} Iω^2. \qquad (6\text{-}19)$$

Ἀπό τίς Ἐξ. (6-18) καί (6-19) προκύπτει

$$mgsημφ = \frac{1}{2} mυ^2 + \frac{1}{2} Iω^2 \qquad (6\text{-}20)$$

καί ἀπ'αὐτή, γιά

$$\omega = \frac{\upsilon}{r} \quad \text{καί} \quad I = \frac{2}{5} mr^2, \tag{6-21}$$

$$mgs\eta\mu\varphi = \frac{1}{2} m\upsilon^2 + \frac{1}{2} \cdot \frac{2}{5} mr^2 \cdot \frac{\upsilon^2}{r^2}$$

ἤ

$$\upsilon = \sqrt{2 \cdot \frac{5}{7} gs\eta\mu\varphi}.$$

Παρατήρηση. Στή περίπτωση πού θά δεχόμασταν ὅτι ἡ σφαίρα δέν στρεφότανε ἀλλά ἐκτελοῦσε μόνον ὀλίσθηση, θά εἴχαμε

$$mgs\eta\mu\varphi = \frac{1}{2} m\upsilon^2$$

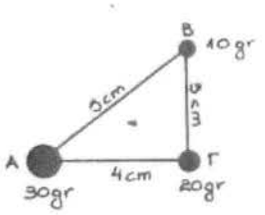
καί

$$\upsilon' = \sqrt{2gs\eta\mu\varphi}$$

δηλαδή

$$\upsilon' > \upsilon.$$

Παράδειγμα 6-2. Τρεῖς μικρές σφαῖρες, πού μποροῦν νά θεωρηθοῦν ὡς ὑλικά σημεῖα, συνδέονται μέ λεπτές στερεές ράβδους, ὅπως δείχνει τό Σχ. 6-6. Ποιά εἶναι ἡ ροπή ἀδράνειας τοῦ συστήματος α) ὡς πρός ἄξονα πού περνάει μέσα ἀπό τό σημεῖο Α, κάθετα πρός τό ἐπίπεδο τοῦ σχήματος, καί β) ὡς πρός ἄξονα πού συμπίπτει μέ τή ράβδο ΒΓ;



Σχ. 6-6

Λύση. α) Τό ὑλικό σημεῖο Α βρίσκεται πάνω στόν ἄξονα, ἄρα ἡ ἀπόστασή του ἀπ'αὐτόν εἶναι μηδέν καί ἑπομένως ἡ ροπή ἀδράνειάς του ὡς πρός τό θεωρούμενο ἄξονα εἶναι μηδέν. Γι'αὐτό ἔχουμε

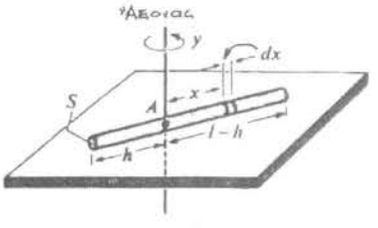
$$I = \Sigma m_i r_i^2 = 10\ gr \times (5^2 cm^2) + 20\ gr \times (4^2 cm^2) = 570\ gr.cm^2.$$

(β) Τά ὑλικά σημεῖα Β καί Γ βρίσκονται καί τά δύο πάνω στό θεωρούμενο ἄξονα. Γι'αὐτό ἔχουμε

$$I = \Sigma m_i r_i^2 = 30\ gr \times (4\ cm^2) = 480\ gr.cm^2.$$

Παρατήρηση. Στό παράδειγμα αὐτό βλέπουμε ὅτι, ἐνῶ ἡ μάζα παραμένει ἡ ἴδια καί στίς δύο περιπτώσεις α) καί β),ἡ ροπή ἀδράνειας τοῦ συστήματος ἀλλάζει σέ κάθε περίπτωση καί αὐτό γιατί ἀλλάζει ὁ ἄξονας περιστροφῆς.

Παράδειγμα 6-3. Νά βρεθεῖ ἡ ροπή ἀδράνειας λεπτῆς ὁμογενοῦς ράβδου, πού ἔχει μάζα m καί μῆκος l, ὡς πρός ἕνα ἄξονα πού περνάει κάθετα πρός τό μῆκος τῆς ράβδου καί ἀπό ἕνα σημεῖο Α, τό ὁποῖο ἀπέχει ἀπόσταση h ἀπό τή μία ἄκρη της.



Σχ. 6-7

Λύση.θεωροῦμε ἕνα στοιχειῶδες κομάτι τῆς ράβδου, μέ μῆκος dx (Σχ. 6-7) καί διατομῆς κάθετης πρός τό μῆκος τῆς ράβδου.Τό κομάτι αὐτό ἀπέχει ἀπό τό σημεῖο Α ἀπόσταση x.. Ἔχουμε

$$dm = \rho dV = \rho S dx = \frac{\rho S l}{l} dx = \frac{m}{l} dx,$$

ὁπότε

$$I_A = \int x^2 dm = \frac{m}{l} \int_{-h}^{l-h} x^2 dx = \frac{m}{l} \cdot \frac{x^3}{3} \Big]_{-h}^{l-h} =$$

$$= \frac{1}{3} m(l^2 - 3lh + 3h^2). \qquad (6\text{-}22)$$

Ἀπ'αὐτή τή γενική ἔκφραση,μποροῦμε νά βροῦμε τή ροπή ἀδράνειας τῆς ράβδου, ὡς πρός ἕναν ἄξονα πού περνάει κάθετα ἀπό ἕνα ὁποιοδήποτε σημεῖο της. Π.χ., ἄν ὁ ἄξονας περνάει ἀπό τό ἀριστερό ἄκρο τῆς ράβδου,ὅπου h=0, προκύπτει

$$I = \frac{1}{3} ml^2.$$

Ἄν ὁ ἄξονας περνάει ἀπό τό δεξιό ἄκρο τῆς ράβδου, ὅπου h=l, τότε

$$I = \frac{1}{3} ml^2.$$

Ἂν ὁ ἄξονας περνάει ἀπό τό κέντρο τῆς ράβδου, ὅπου $h = \frac{1}{2}$ ἔχουμε

$$I = \frac{1}{12} ml^2.$$

6-7 Θεμελιώδης νόμος τῆς στροφικῆς κινήσεως

Ὅπως μιά δύναμη, πού ἐξασκεῖται σ' ἕνα ὑλικό σημεῖο, ἔχει σάν ἀποτέλεσμα τήν ἐπιτάχυνση, ἔτσι καί μία ροπή, πού ἐξασκεῖται σ' ἕνα στερεό σῶμα, στρεπτό γύρω ἀπό ἀκλόνητο ἄξονα, ἔχει σάν ἀποτέλεσμα γωνιακή ἐπιτάχυνση. Μεταξύ τῆς ροπῆς $\vec{M}$ - αἴτιο - καί τῆς γωνιακῆς ἐπιταχύνσεως $\vec{\alpha}$ - ἀποτέλεσμα - ἰσχύει ἡ σχέση

$$\boxed{\vec{M} = I\vec{\alpha}',} \qquad (6\text{-}23)$$

ὅπου I ἡ ροπή ἀδράνειας τοῦ σώματος ὡς πρός τόν ἄξονα περιστροφῆς.

Ἡ σχέση αὐτή ἀποτελεῖ τή θεμελιώδη ἐξίσωση (νόμο) τῆς στροφικῆς κινήσεως.

Ἀπό τή διερεύνηση τῆς Ἐξ. (6-23) προκύπτουν τά ἑξῆς:

α) Γιά $\vec{M} = 0$ προκύπτει καί $\vec{\alpha}' = 0$.

Αὐτό σημαίνει ὅτι ἡ γωνιακή ταχύτητα τοῦ στρεφόμενου σώματος θά διατηρεῖται σταθερή. Ἐπομένως, ἄν σ' ἕνα σῶμα, πού μπορεῖ νά στραφεῖ γύρω ἀπό σταθερό ἄξονα, δέν ἐξασκοῦνται ροπές ἤ ἐξασκοῦνται ροπές ἀλλά ἡ συνισταμένη τους εἶναι ἴση μέ τό μηδέν, τότε, ἄν μέν τό σῶμα ἀρχικά δέν περιστρέφονταν θά ἐξακολουθεῖ νά μή περιστρέφεται, ἄν δέ περιστρέφονταν μέ σταθερή γωνιακή ταχύτητα, θά ἐξακολουθεῖ νά περιστρέφεται μέ τήν ἴδια γωνιακή ταχύτητα. Τά πιό πάνω ἰσχύουν καί ἀντίστροφα.

β) Γιά $\vec{M}$ = σταθ. παίρνουμε καί $\vec{\alpha}'$ = σταθ.

6-8 Στροφορμή

Μέ βάση τήν Ἐξ. (4-25) ὁρίσαμε τή στροφορμή ὑλικοῦ σημείου. Σέ πλήρη ἀναλογία πρός τό ὑλικό σημεῖο, ὁρίζουμε σάν στροφορμή $\vec{L}$ ἑνός σώματος, πού μπορεῖ νά περιστρέφεται γύρω ἀπό σταθερό ἄξονα, τό διανυσματικό ἄθροισμα τῶν στροφορμῶν ὅλων τῶν σημείων τοῦ σώματος.

Στήν προκειμένη περίπτωση, ἄν ληφθεῖ ὑπόψη ὅτι τά διανύσματα τῶν στροφορμῶν ὅλων τῶν ὑλικῶν σημείων τοῦ σώματος ἔχουν τήν ἴδια διεύθυνση, τό δ ι α ν υ σ μ α τ ι κ ό ἄθροισμα γίνεται ἕνα ἀριθμητικό ἄθροισμα. Δηλαδή ἰσχύει

$$L = \Sigma r_i m_i v_i, \tag{6-24}$$

ἀπό τήν ὁποία καί τήν $v_i = \omega_i r_i$ προκύπτει

$$L = \Sigma m_i \omega_i r_i^2 \tag{6-25}$$

ἤ, ἐπειδή ω_i = σταθ. καί $I = \Sigma m_i r_i^2$,

$$L = I.\omega. \tag{6-26}$$

Ἡ Ἐξ. (6-26) διανυσματικά γράφεται

$$\boxed{\vec{L} = I.\vec{\omega},} \tag{6-27}$$

ὁπότε ἡ Ἐξ. (6-23) γίνεται

$$\vec{M} = I \frac{d\vec{\omega}}{dt} = \frac{d}{dt}(I\vec{\omega})$$

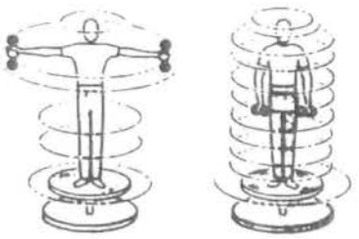
ἤ

$$\boxed{\vec{M} = \frac{d\vec{L}}{dt}.} \tag{6-28}$$

Ἡ σχέση αὐτή ἀποτελεῖ τή γ ε ν ι κ ό τ ε ρ η μ ο ρ φ ή τ ῆ ς θ ε μ ε λ ι ώ δ ο υ ς ἐ ξ ι σ ώ σ ε ω ς τ ῆ ς σ τ ρ ο φ ι κ ῆ ς κ ι ν ή σ ε ω ς.

Ἀπ'τήν Ἐξ. (6-28), γιά $\vec{M}=0$ προκύπτει καί $\frac{d\vec{L}}{dt} = 0$. Τοῦτο σημαίνει ὅτι γιά $\vec{M}=0$ ἡ στροφορμή τοῦ σώματος διατηρεῖται σταθερή. Ἐφαρμογή αὐτοῦ ἔχομε στό πιό κάτω πείραμα: Ἄνθρωπος, πού κρατάει στά χέρια του βάρη (Σχ. 6-8), στέκεται, μέ τά χέρια του τεντωμένα ὁριζόντια, πάνω σέ βάθρο, τό ὁποῖο μπορεῖ νά περιστρέφεται γύρω ἀπό κατακόρυφο ἄξονα. Ὁ ἄνθρωπος αὐτός μέ μία μικρή ὥθηση ἀρχίζει νά περιστρέφεται μέ σταθερή γωνιακή ταχύτητα ω_1, ἀφοῦ $\vec{M}=0$. Ἔστω δέ I_1 ἡ ροπή ἀδράνειάς του σ'αὐτήν τήν περί-

πτωση. Ἂν τώρα ὁ ἄνθρωπος,καθώς στρέφεται, πλησιάσει τά βάρη πρός τό σῶμα του, ἡ ροπή ἀδράνειάς του θά γίνει μικρότερη, ἔστω I_2. Ἐπειδή ὅμως ἡ στροφορμή πρέπει νά παραμείνη σταθερή, καθόσον $\vec{M}=0$,ὁ στρεφόμενος ἄνθρωπος θά ἀποκτήσει νέα γωνιακή ταχύτητα ω_2,μεγαλύτερη τῆς ω_1, ὥστε νά ἰσχύει



Σχ. 6-8. Διατήρηση τῆς στροφορμῆς.

$$I_1\omega_1 = I_2\omega_2.$$

6-9 Ὥθηση τῆς ροπῆς

Ἀπό τήν Ἐξ. (6-28), μέ ὁλοκλήρωση ἀπό τή χρονική στιγμή $t = t_o$ μέχρι τή χρονική στιγμή $t = t_1$, ἔχουμε

$$\int_{t_o}^{t_1} \vec{M}dt = \vec{L}_1 - \vec{L}_o. \qquad (6\text{-}29)$$

Ἡ παράσταση

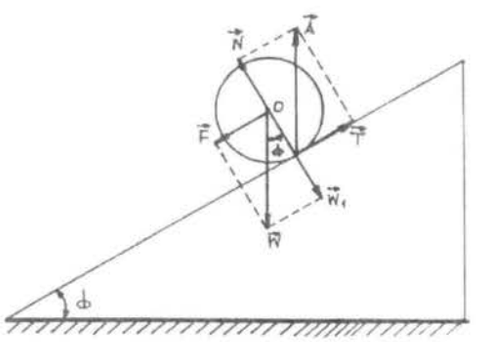
$$\int \vec{M}dt$$

ὀνομάζεται ὥ θ η σ η ρ ο π ῆ ς καί εἶναι ἀνάλογη πρός τήν ὥθηση δυνάμεως $\int \vec{F}dt$.

Παράδειγμα 6-4. Ἕνας συμπαγής κύλινδρος μέ ἀκτίνα r κυλίεται χωρίς ὀλίσθηση κατά μῆκος ἑνός κεκλιμένου ἐπιπέδου, πού ἔχει γωνία κλίσεως φ. Νά ὑπολογιστεῖ ἡ ἐπιτάχυνση τοῦ κέντρου τοῦ κυλίνδρου.Δίνεται ὅτι ἡ ροπή ἀδράνειας τοῦ κυλίνδρου ὡς πρός τόν ἄξονά του εἶναι $I = \frac{1}{2} mr^2$.

Λύση. Στόν κύλινδρο (Σχ. 6-9) ἀναπτύσσονται οἱ ἑξῆς δυνάμεις: Τό βάρος $\vec{w}$ καί ἡ ἀντίδραση $\vec{A}$ ἀπό τό ἐπίπεδο, ἡ ὁποία ἀναλύεται στή κάθετη συνιστῶσα $\vec{N}$ καί στήν τριβή $\vec{T}$.

Ἡ τριβή $\vec{T}$ πρέπει νά ὑπάρχει ὁπωσδήποτε γιά νά μπορεῖ ὁ κύλινδρος νά ἐκτελέσει κύλιση· καί τοῦτο γιατί,γιά νά ἐκτελέσει ὁ κύλιν-



Σχ. 6-9

12-7 Νόμος τοῦ Stokes

Ἡ μορφή τῆς ροῆς ἑνός πραγματικοῦ ρευστοῦ γύρω ἀπό μιά σφαίρα ἐξαρτᾶται ἀπό τήν ταχύτητα τοῦ ρευστοῦ. Σέ μικρές ταχύτητες ἡ ροή εἶναι στρωτή, ἐνῶ σέ μεγαλύτερες εἶναι τυρβώδης.

Στήν περίπτωση τῆς στρωτῆς ροῆς γύρω ἀπό σφαίρα, ἡ ἀντίσταση R, πού ἀναπτύσσεται, δίνεται ἀπό τή σχέση

$$\boxed{R = 6\pi\eta r u,} \qquad (12\text{-}16)$$

ὅπου η εἶναι ὁ συντελεστής ἰξώδους τοῦ ρευστοῦ r ἡ ἀκτίνα τῆς σφαίρας καί υ ἡ σχετική ταχύτητα τοῦ ὑγροῦ ὡς πρός τή σφαίρα. Ἡ πιό πάνω σχέση ἐκφράζει τό ν ό μ ο τ ο ῦ S t o k e s. Ἐδῶ πρέπει νά σημειωθεῖ ἰδιαίτερα ὅτι, ὅπως βλέπουμε στήν Ἐξ. (12-16), στή στρωτή ροή ἡ ἀντίσταση εἶναι ἀνάλογη μέ τή πρώτη δύναμη τῆς ταχύτητας.

12-8 Ἀντίσταση στήν τυρβώδη ροή

Ἡ ἀντίσταση R, πού ἀναπτύσσεται σέ σῶμα ὁποιουδήποτε σχήματος μέσα σέ τυρβώδη ροή, βρίσκεται ἀπό τή σχέση

$$\boxed{R = C_{\alpha\nu} A_{\mu\epsilon\tau} \frac{\rho}{2} u^2,} \qquad (12\text{-}17)$$

ὅπου $C_{\alpha\nu}$ εἶναι μία σταθερή (καθαρός ἀριθμός), ἡ ὁποία ἐξαρτᾶται ἀπό τό

σχῆμα τοῦ σώματος καί μάλιστα ἀπό τό πίσω μέρος του, καί ἡ ὁποία ὀνομάζεται σ υ ν τ ε λ ε σ τ ή ς ἀ ν τ ι σ τ ά σ ε ω ς. Ἐξαιρετικά μικρό συντελεστή ἀντιστάσεως ἔχει τό ἰχθυοειδές σχῆμα, πού λέγεται καί ἀ ε ρ ο δ υ ν α μ ι κ ό. Τό $A_{μετ}$ εἶναι τό ἐμβαδό τῆς μέγιστης διατομῆς τοῦ σώματος, πού εἶναι κάθετη στή διεύθυνση τοῦ ρευστοῦ, καί ὀνομάζεται μ ε τ ω π ι κ ή ἐ π ι φ ά ν ε ι α. Τό υ εἶναι ἡ σχετική ταχύτητα τοῦ ρευστοῦ ὡς πρός τό σῶμα καί τό ρ ἡ πυκνότητα τοῦ ρευστοῦ.

Ἡ Ἐξ. 12-17 ἀποδείχνεται θεωρητικά καί πειραματικά καί ἔχει μεγάλη πρακτική σημασία, γιατί μέ αὐτή ὑπολογίζεται ἡ ἀντίσταση ὀχημάτων, μέ μεγάλη ταχύτητα (αὐτοκινήτων, ἀεροπλάνων κ.λ.π.).

Παράδειγμα 12-2. Ἕνα ἀεροπλάνο κινεῖται εὐθύγραμμα καί μέ σταθερή ταχύτητα. Στή συνέχεια διπλασιάζεται ἡ ταχύτητά του. α) Νά βρεθεῖ ἡ μεταβολή τῆς ἰσχύος τοῦ κινητήρα τοῦ ἀεροπλάνου καί β) ἄν ἡ ἰσχύς τοῦ κινητήρα διπλασιαστεῖ, πόση ἡ μεταβολή τῆς ταχύτητας τοῦ ἀεροπλάνου;

Λύση. α) Ἐφόσον ἡ ταχύτητα τοῦ ἀεροπλάνου διατηρεῖται σταθερή ἔχουμε

$$P = Fu.$$

Ἀλλά ἡ F εἶναι ἴση μέ τήν R, πού παίρνουμε ἀπό τήν Ἐξ. (12-17). Ἑπομένως, στήν ἀρχή ἰσχύει

$$P_1 = C_{αν} A_{μετ} \frac{ρ}{2} u_1^3 \qquad (12\text{-}18)$$

καί μετά, ὅπου $u_2 = 2u_1$,

$$P_2 = C_{αν} A_{μετ} \frac{ρ}{2} (2u_1)^3$$

ἤ

$$P_2 = C_{αν} A_{μετ} \frac{ρ}{2} 8u_1^3. \qquad (12\text{-}19)$$

Ἀπό τίς Ἐξ. (12-18) καί (12-19), μέ διαίρεση κατά μέλη, ἔχουμε

$$\frac{P_2}{P_1} = 8$$

ἤ

$P_2 = 8P_1$.

β) Ἄν τήν Ἐξ. (12-17) ἐφαρμόσουμε στήν

$P_2' = 2P_1'$

θά πάρουμε

$$C_{αν}A_{μετ}\frac{\rho}{2}\upsilon_2^3 = 2\,C_{αν}A_{μετ}\frac{\rho}{2}\upsilon_1^3$$

ἤ

$$\upsilon_2 = \upsilon_1\sqrt[3]{2}\,.$$

12-9 Ἀριθμός Reynolds

Πειραματικές παρατηρήσεις ἔδειξαν ὅτι ἡ στρωτή ροή ἀλλάζει καί γίνεται τυρβώδης, ἄν ἡ ταχύτητά της ξεπεράσει μία ὁρισμένη τιμή. Ἡ τιμή αὐτή ὀνομάζεται κ ρ ί σ ι μ η τ α χ ύ τ η τ α $\upsilon_{κρ}$. Ἡ κρίσιμη ταχύτητα ὁρίζει ἕνα ἀριθμό, ὁ ὁποῖος χαρακτηρίζει τή ροή καί ὁ ὁποῖος ὀνομάζεται κ ρ ί σ ι μ ο ς ἀ ρ ι θ μ ό ς R e y n o l d s N_R.

Γιά ροή μέσα σέ σωλήνα, ὁ κρίσιμος ἀριθμός Reynolds ὁρίζεται ἀπό τή σχέση

$$N_R = \frac{\rho\upsilon_{κρ}.D}{n}\,, \qquad (12\text{-}20)$$

ὅπου ρ εἶναι ἡ πυκνότητα τοῦ ρευστοῦ, n ὁ συντελεστής ἰξώδους, $\upsilon_{κρ}$ ἡ κρίσιμη ταχύτητα καί D μία ἀπό τίς γραμμικές διαστάσεις τῆς διατομῆς τοῦ σωλήνα. Π.χ., ἄν ἡ διατομή εἶναι κυκλική τότε τό D παριστάνει τήν ἀκτίνα r τῆς διατομῆς, ἄν ἡ διατομή εἶναι τετραγωνική τό D παριστάνει τό μῆκος l μιᾶς ἀπό τίς πλευρές της κ.ο.κ.

Ἀπό τήν Ἐξ. (12-20) παρατηροῦμε ὅτι, ἄν ἡ ταχύτητα υ τῆς ροῆς εἶναι μικρότερη ἀπό τήν $\upsilon_{κρ}$, τότε, ὁ ἀριθμός $\frac{\rho\upsilon D}{n}$, πού λέγεται ἀ ρ ι θ μ ό ς R e y n o l d s, εἶναι μικρότερος τοῦ κρίσιμου. Ἔτσι μποροῦμε νά κρίνουμε ἄν μία ροή εἶναι στρωτή ἤ ὄχι, ἀρκεῖ ἀπό τά δεδομένα υ, ρ, D καί n νά ὑπολογίσουμε τόν ἀριθμό Reynolds καί νά τόν συγκρίνουμε μέ τόν

κρίσιμο ἀριθμό Reynolds. Ὁπότε, ἄν

$$\frac{\rho u D}{\eta} < N_R,$$

τότε ἡ ροή θά εἶναι στρωτή, καί ἄν

$$\frac{\rho u D}{\eta} \geq N_R,$$

τότε ἡ ροή θά εἶναι τυρβώδης.

Ὁ κρίσιμος ἀριθμός Reynolds βρίσκεται πειραματικά, εἶναι καθαρός ἀριθμός, ὅπως φαίνεται καί ἀπό τή σχέση ὁρισμοῦ του καί ἔχει ἰδιαίτερη σημασία στά προβλήματα ροῆς. Ἀποδείχνεται ὅτι ρεύματα (ροές), στά ὁποῖα ἀντιστοιχεῖ ὁ ἴδιος ἀριθμός Reynolds, εἶναι ὅμοια.

Ὅλα τά πειράματα ἔδειξαν ὅτι, ὅταν ὁ ἀριθμός Reynolds εἶναι μικρότερος περίπου ἀπό 2000, ἡ ροή εἶναι στρωτή. Ἔτσι π.χ., νερό στούς $20^{\circ}C$, ὅπου $\eta = 0{,}01$ dynsec/cm^2, πού ρέει μέσα σέ ἕνα σωλήνα κυκλικῆς διατομῆς ἀκτίνας 1 cm, ἔχει στρωτή ροή, ὅταν

$$\frac{\rho u D}{\eta} \leq 2000$$

ἤ ὅταν

$$u \leq \frac{2000 \times 0{,}01}{1 \times 1} \frac{cm}{sec} = 20 \frac{cm}{sec}.$$

Ἐξάλλου, ἄν θεωρήσουμε, τώρα, ἀέρα στήν ἴδια θερμοκρασία τῶν $20^{\circ}C$, κινούμενο μέσα στόν ἴδιο σωλῆνα, μέ τήν ἴδια ταχύτητα τῶν 20 cm/sec, ἐπειδή εἶναι $\rho = 0{,}0013$ gr/cm^3 καί $\eta = 181 \times 10^{-6}$ dyn.sec/cm^2 (βλ. Πίνακα 12-1), θά ἔχουμε

$$\frac{\rho u D}{\eta} = \frac{0{,}0013 \times 20 \times 1}{181 \times 10^{-6}} = 144.$$

Ἀπό τό ἀποτέλεσμα αὐτό βλέπουμε ὅτι ὁ ἀριθμός Reynolds στήν πιό πάνω ροή τοῦ ἀέρα εἶναι κατά πολύ μικρότερος τοῦ 2000. Ἄρα ἡ ροή αὐτή θά εἶναι ὁπωσδήποτε στρωτή. Ἀκόμη, ἀπό τήν Ἐξ. (12-20) προκύπτει ἡ κρίσιμη ταχύτητα

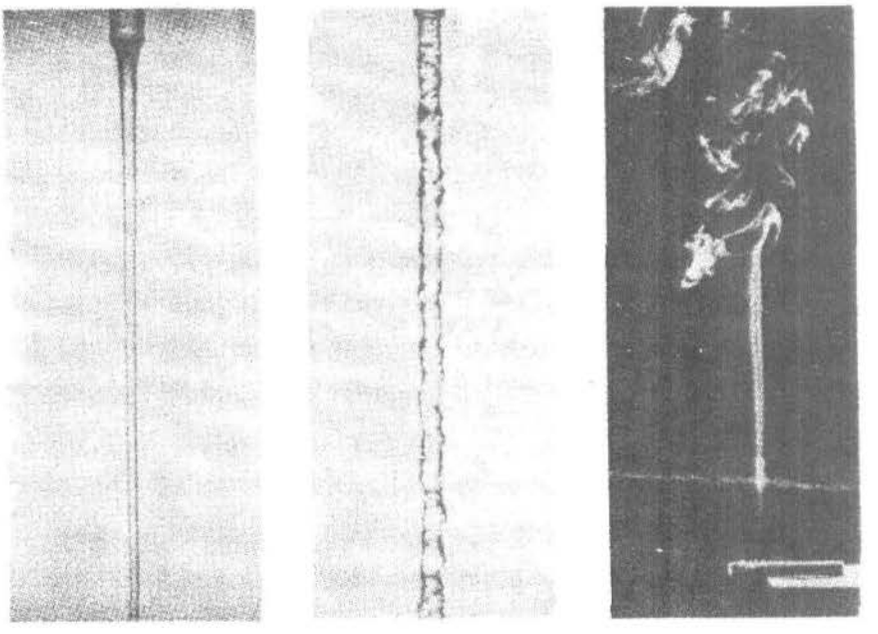
$$v_{\kappa\rho} = \frac{N_R \eta}{\rho D} ,$$

πού, γιά τήν πιό πάνω περίπτωση, εἶναι

$$v_{\kappa\rho} = \frac{2 \times 10^3 \times 181 \times 10^{-6}}{1,3 \times 10^{-3} \times 1} = 278,46 \frac{cm}{sec} .$$

Δηλαδή στή θεωρούμενη περίπτωση, ὅταν ὁ ἀέρας κινεῖται μέ ταχύτητα μέχρι 278,46 cm/sec ἡ ροή εἶναι στρωτή, ἐνῶ, ὅταν ὁ ἀέρας κινεῖται μέ μεγαλύτερη ταχύτητα ἡ ροή εἶναι τυρβώδης.

Ἡ διαφορά μεταξύ στρωτῆς καί τυρβώδους ροῆς φαίνεται χαρακτηριστικά στό Σχ. (12-9). Στό (α) ἡ ροή τοῦ νεροῦ εἶναι στρωτή, στό (β) ἡ ροή τοῦ νεροῦ εἶναι τυρβώδης καί στό (γ) ἡ ροή καπνοῦ ἀπό τσιγάρο εἶναι στήν ἀρχή στρωτή καί μετά τυρβώδης.



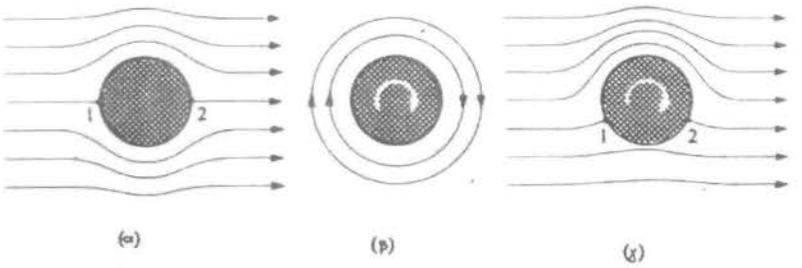
(α) (β) (γ)

Σχ. 12-9. (α) Στρωτή ροή. (β) Τυρβώδης ροή. (γ). Πρῶτα στρωτή μετά τυρβώδης.

12-10 Δυναμική ἄνωση

Ὅταν ἕνα ἰδανικό ρευστό ρέει, κατά μία εὐθύγραμμη διεύθυνση, γύρω

ἀπό ἕναν ἀκίνητο κύλινδρο, οἱ ρευματικές γραμμές θά κατανέμονται ὅπως δείχνει τό Σχ. 12-10α. Ἄν ἐξάλλου τό ρευστό ἐκτελεῖ μόνο περιφορά γύρω ἀπό τόν κύλινδρο (πράγμα πού ἐπιτυγχάνεται μέ περιστροφή τοῦ κυλίνδρου μέσα σέ ἀκίνητο ρευστό), θά προκύψουν οἱ ρευματικές γραμμές τοῦ Σχ. 12-10β. Ἄν, τώρα, τό ρευστό ἐκτελεῖ ταυτόχρονα καί τίς δύο κινήσεις, ἡ συνισταμένη κίνηση θά προκύψει ἀπό τήν σύνθεση τῶν δύο συνιστωσῶν κινήσεων, ἡ δέ μορφή τῶν ρευματικῶν γραμμῶν θά ἐμφανίζεται στό Σχ. 12-10γ. Ἐδῶ παρατηροῦμε ὅτι οἱ ρευματικές γραμμές σέ ἄλλες περιοχές ἐμφανίζουν συμπύκνωση (πάνω), σέ ἄλλες ἀραίωση (κάτω). Συμπύκνωση ἔχουμε στίς περιοχές ὅπου ἡ ταχύτητα τῆς ροῆς καί ἡ ταχύτητα ἀπό τήν περιφορά ἔχουν τήν ἴδια φορά, ἐνῶ ἀραίωση ἔχουμε ἐκεῖ ὅπου οἱ ταχύτητες αὐτές ἔχουν ἀντίθετες φορές.

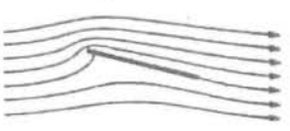


Σχ. 12-10

Ἡ διαφορά στίς ταχύτητες στήν πάνω καί κάτω ἐπιφάνεια τοῦ κυλίνδρου δημιουργεῖ, σύμφωνα μέ τό νόμο τοῦ Bernoulli, διαφορά πιέσεως, μέ ἀποτέλεσμα τήν ἐμφάνιση δυνάμεως, κάθετης στή διεύθυνση τῆς ροῆς. Ἡ δύναμη αὐτή ὀνομάζεται δ υ ν α μ ι κ ή ἄ ν ω σ η, τό δέ φαινόμενο πού περιγράψαμε πιό πάνω λέγεται φ α ι ν ό μ ε ν ο M a g n u s. Τό φαινόμενο αὐτό, πού περιγράψαμε στά ἰδανικά ρευστά, δίνει τό ἴδιο περίπου ἀποτέλεσμα καί στά πραγματικά. Στό φαινόμενο Magnus ὀφείλεται καί ἡ χαρακτηριστική τροχιά πού κάνει μιά ἐκσφενδονιζόμενη σφαῖρα, ἡ ὁποία ταυτόχρονα περιστρέφεται (ποδόσφαιρο, ἀντισφαίριση κ.λ.π.).

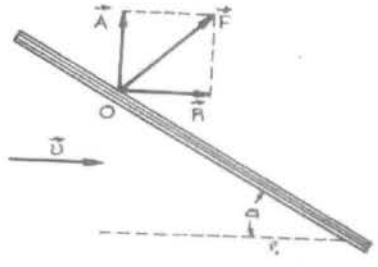
12-11 Δυναμική ἄνωση στό ἀεροπλάνο

Στό Σχ. 12-11 ἐμφανίζονται οἱ ρευματικές γραμμές πραγματικοῦ ρευ-



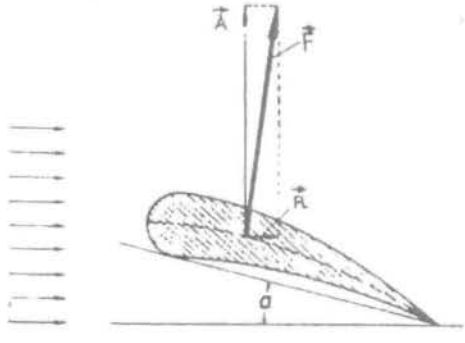
Σχ. 12-11

στοῦ γύρω ἀπό ἐπίπεδη πλάκα, πού σχηματίζει μικρή γωνία μέ τή διεύθυνση τῆς ροῆς. Παρατηροῦμε ὅτι, λόγω τῆς κατανομῆς τῶν δυναμικῶν γραμμῶν τοῦ ρευστοῦ γύρω ἀπό τήν πλάκα, στό κάτω μέρος της ὑπάρχει παντοῦ ὑπερπίεση, ἐνῶ στό πάνω ὑποπίεση. Ἀποτέλεσμα τούτου εἶναι νά ἐμφανίζεται στή πλάκα, ἐκτός ἀπό τή ροπή καί μία δύναμη,(Σχ. 12-12) πλάγια πρός τήν κατεύθυνση τῆς ροῆς, περίπου κάθετη στήν πλάκα καί διερχόμενη ἀπό ἕνα σημεῖο Ο, πού βρίσκεται πλησιέστερα στό μπροστινό ἄκρο τῆς πλάκας. Ἡ δύναμη αὐτή $\vec{F}$,πού λέγεται **ἀεροδύναμη**, ἀναλύεται σέ δύο συνιστῶσες, μία $\vec{A}$ κάθετη στή διεύθυνση τῆς ροῆς, πού ὀνομάζεται **δυναμική ἄνωση**, καί τήν $\vec{R}$ παράλληλη στή ροή,πού ὀνομάζεται **δυναμική ἀντίσταση**.



Σχ. 12-12

Στήν πλάκα ἡ δυναμική ἀντίσταση εἶναι μεγάλη σέ σχέση πρός τή δυναμική ἄνωση. Σημαντική βελτίωση ἔχουμε στήν **πτέρυγα** τοῦ ἀεροπλάνου,ὅπου ἡ ἀντίσταση εἶναι πολύ μικρότερη (Σχ. 12-13). Ἡ πτέρυγα τοῦ ἀεροπλάνου διαμορφώνεται κατάλληλα, ὥστε μία ἐγκάρσια τομή της νά ἔχει σχῆμα ἰχθυοειδές. Ἡ μορφή τῶν ρευματικῶν γραμμῶν τοῦ ρευστοῦ γύρω ἀπό μιά πτέρυγα ἀεροπλάνου φαίνονται στό Σχ. 12-14. Παρατηροῦμε ὅτι, ὅταν αὐξάνεται ἡ γωνία, πού σχηματίζει ἡ διεύθυνση τῆς ροῆς μέ τή διεύθυνση τῆς πτέρυγας (**γωνία πρόσβολῆς**), προκαλεῖται με-

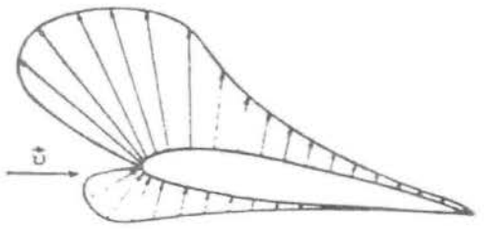


Σχ. 12-13. Τομή πτέρυγας ἀεροπλάνου, $\vec{F}$=ἀεροδύναμη, $\vec{A}$ = δυναμική ἄνωση,$\vec{R}$=δυναμική ἀντίσταση, α = γωνία προσβολῆς.

(α) (β) (γ)

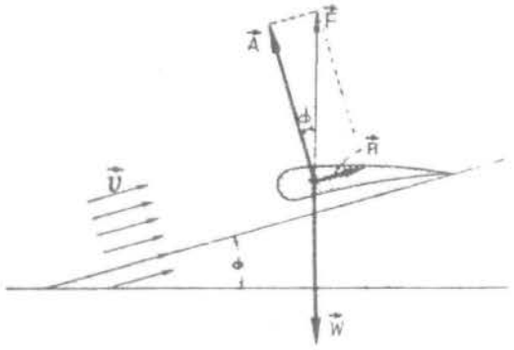
Σχ. 12-14. Κατανομή τῶν ρευματικῶν γραμμῶν ρευστοῦ γύρω ἀπό πτέρυγα ἀεροπλάνου, γιά τρεῖς διαφορετικές γωνίες προσβολῆς.

γαλύτερη διατάραξη τῆς μορφῆς τῶν ρευματικῶν γραμμῶν. Ἡ μορφή αὐτή τῶν ρευματικῶν γραμμῶν δημιουργεῖ τέτοια κατανομή τῶν πιέσεων γύρω ἀπό τήν πτέρυγα, ὥστε στήν πάνω ἐπιφάνεια νά ἐμφανίζονται ὑποπιέσεις,δηλαδή πιέσεις μικρότερες τῆς ἀτμοσφαιρικῆς καί στήν κάτω ἐπιφάνεια ὑπερπιέσεις, δηλαδή πιέσεις μεγαλύτερες τῆς ἀτμοσφαιρικῆς. Στό Σχ. 12-15 ἐμφανίζεται ἡ κατανομή τῶν πιέσεων γύρω ἀπό τήν πτέρυγα ἀεροπλάνου.Στό διάγραμμα αὐτό,τό μέν μῆκος τῶν βελῶν ἔχει παρθεῖ ἀνάλογο πρός τήν ὑποπίεση ἤ τήν ὑπερπίεση πού ἐπικρατεῖ στό ἀντίστοιχο σημεῖο, ἡ δέ φορά τους, πρός τήν ἐπιφάνεια ἄν ἀντιστοιχοῦν σέ ὑπερπιέσεις, ἀπό τήν ἐπιφάνεια στήν ἀντίθετη περίπτωση.



Σχ. 12-15. Κατανομή τῶν πιέσεων γύρω ἀπό πτέρυγα ἀεροπλάνου. Ἡ φορά τῶν βελῶν δείχνει ὑπερπιέσεις στήν κάτω ἐπιφάνεια καί ὑποπιέσεις στήν πάνω.

Ἡ πιό πάνω κατανομή τῶν πιέσεων γύρω ἀπό τήν πτέρυγα ἀεροπλάνου ἔχει σάν ἀποτέλεσμα, τήν ἐμφάνιση ἀεροδυνάμεως πάνω σ'αὐτή (Σχ. 12 - 16).



Σχ. 12-16

Σχ. 12-17

Ἡ ἀεροδύναμη $\vec{F}$ ἀναλύεται στήν δυναμική ἄνωση $\vec{A}$ καί στήν δυναμική ἀντίσταση $\vec{R}$. Οἱ δυνάμεις $\vec{A}$ καί $\vec{R}$ ἐκφράζονται μέ βάση τήν Ἐξ.(12-17),μέ μόνη τή διαφορά ὅτι σ'αὐτή τήν περίπτωση τό ἐμβαδό τῆς μετωπικῆς ἐπιφάνειας ἀντικατασταίνεται ἀπό τό ἐμβαδό τῆς μιᾶς ἀπό τίς δύο ἐπιφάνειες τῆς πτέρυγας (φ έ ρ ο υ σ α έ π ι φ ά ν ε ι α). Δηλαδή ἰσχύει

$$A = C_A \frac{\rho v^2}{2} bl$$

καί

$$R = C_R \frac{\rho u^2}{2} bl,$$

ὅπου b εἶναι τό μῆκος τῆς χορδῆς τῆς πτέρυγας (Σχ. 12-17), l τό πλάτος της, C_A ὁ συντελεστής ἀνώσεως καί C_R ὁ συντελεστής ἀντιστάσεως. Οἱ δύο αὐτοί συντελεστές ἐξαρτῶνται ἀπό τή γωνία προσβολῆς α καί τόν ἀριθμό Reynolds τῆς ροῆς.

Ἡ γωνία φ πού σχηματίζεται μεταξύ τῶν δυνάμεων $\vec{F}$ καί $\vec{A}$ λέγεται γωνία ὀλισθήσεως, ἡ δέ ἐφαπτομένη της ε ἀριθμός ὀλισθήσεως. Εἶναι δέ (Σχ. 12-16)

$$\varepsilon\varphi\varphi = \frac{R}{A} = \varepsilon.$$

Σέ μιά νεώτερου τύπου πτέρυγα, ὁ ἀριθμός ὀλισθήσεως εἶναι περίπου $\frac{1}{25}$ ἤ καί μικρότερος.

Προβλήματα

12-1. Μέσα σέ ὁριζόντιο σωλήνα ρέει νερό. Ὁ σωλήνας ἔχει στήν ἀρχή διατομή $A_1 = 4\ cm^2$, ἔπειτα ὅμως στενεύει καί ἡ διατομή του γίνεται $A_2 = 1\ cm^2$. Στίς δύο αὐτές τομές τοῦ σωλήνα εἶναι στερεωμένοι κατακόρυφοι λεπτοί σωλῆνες καί στόν πρῶτο ἀπό αὐτούς τό νερό σχηματίζει στήλη ὕψους $h_1 = 15\ cm$. Ἄν ἡ ταχύτητα τοῦ νεροῦ στή μικρή τομή εἶναι 80 cm/sec, νά βρεθεῖ πόσο εἶναι τό ὕψος h_2 τῆς στήλης τοῦ νεροῦ στόν ἄλλο κατακόρυφο σωλήνα. (Ἀπ. $h_2 = 11,94\ cm$).

12-2. Δοχεῖο περιέχει νερό σέ ὕψος 1m καί ἀέρα μέ πίεση 0,5 At μεγαλύτερη ἀπό τήν ἀτμοσφαιρική. Τό νερό τρέχει ἀπό μιά τρύπα πού βρίσκεται στόν πυθμένα τοῦ δοχείου καί ἔχει ἐμβαδό $A = 5\ cm^2$. Νά ὑπολογιστεῖ ἡ ταχύτητα ἐκροῆς, τή στιγμή πού ἀρχίζει νά βγαίνει νερό ἀπό τήν τρύπα, καθώς καί ἡ δύναμη πού ἐξασκεῖ ἡ φλέβα στό δοχεῖο. (Ἀπ. 1050 cm / sec, $5,5 \times 10^6$).

12-3. Μιά σταγόνα ἑνός σύνεφου, γιά νά πέσει στό ἔδαφος μέ κίνηση ὁμαλά ἐπιταχυνόμενη, θέλει χρόνο 4 min. Ἄλλη ἴδια σταγόνα πρίν ξεκινήσει διασπᾶται σέ 27 ὅμοια σταγονίδια. Νά βρεθεῖ ὁ χρόνος πού θέλουν τά σταγονίδια γιά νά φτάσουν στό ἔδαφος, ἄν δεχτοῦμε ὅτι ἡ ἀντίσταση τοῦ