Chapter 1: Hardware and Software Overview

In Fall 2019, ROBOTIS released the ENGINEER Kit 1 (https://emanual.robotis.com/docs/en/edu/engineer/kit1/#introduction) and followed up in Spring 2020 with the release of the ENGINEER Kit 2 (https://emanual.robotis.com/docs/en/edu/engineer/kit2_introduction/). Both kits use a new Hardware Controller named CM-550 which is based on an ARM Cortex-M4 microcontroller clocking at 168 Mhz along with 1024 KB of Flash Memory. This larger memory allows 40.5 KB to be allocated to its TASK section and increases the number of TASK Variables to 200. The higher clock rate and increased memory also allows the embedding of a MicroPython engine inside the CM-550.

A new type of Dynamixel was also premiered: the 2XL430-W250-T containing two servo motors with their axles mounted perpendicular to each other allowing a more compact design of jointed linkages (http://emanual.robotis.com/docs/en/dxl/x/2xl430-w250/). The ENGINEER kits can also use the single-servo version XL430-W250-T (http://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#control-table).

Let us have a closer look at the main features of the hardware and software systems for the ENGINEER kits.

1.1 CM-550

The CM-550 supports only Protocol 2 Dynamixels (http://emanual.robotis.com/docs/en/dxl/protocol2/), thus the older AX-12 and AX-18 actuators from the BIOLOID series are not compatible with this Hardware Controller (3-pin JST connectors are now used instead of the old 3-pin Molex connectors) (Thai, 2017). The XL-320 actuator unfortunately is also not compatible with the CM-550, even though it is a Protocol 2 Dynamixel, because it operates at 7.4 V while the CM-550 operates at 11-12 V.

The CM-550 has 6 X-series Dynamixel ports (3-pin) and 5 GPIO ports (5-pin) (http://emanual.robotis.com/docs/en/parts/controller/cm-550/#specifications). These 5-pin ports allow the use of small sensors and actuators developed for the DREAM and MINI systems (http://emanual.robotis.com/docs/en/parts/controller/controller_compatibility/#parts) (Thai, 2018; Thai, 2020).

The CM-550 retains the buzzer/microphone feature from the older CM-510/CM-530 series, but it has a new feature not found in the older series: an embedded 3-axis accelerometer/gyroscope including a data processor module to generate Roll, Pitch and Yaw angles.

In the communications hardware area, it has one Micro USB port and one 4-pin UART port, and surprisingly an embedded BT-410 module (see Fig. 1.1):

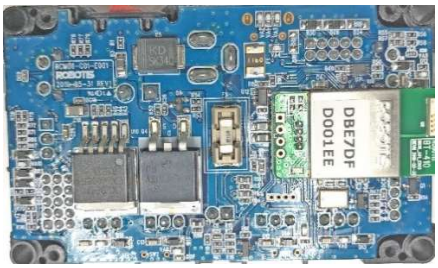


Fig. 1.1 Bottom view of CM-550 showing the embedded BT-410 module on the right.

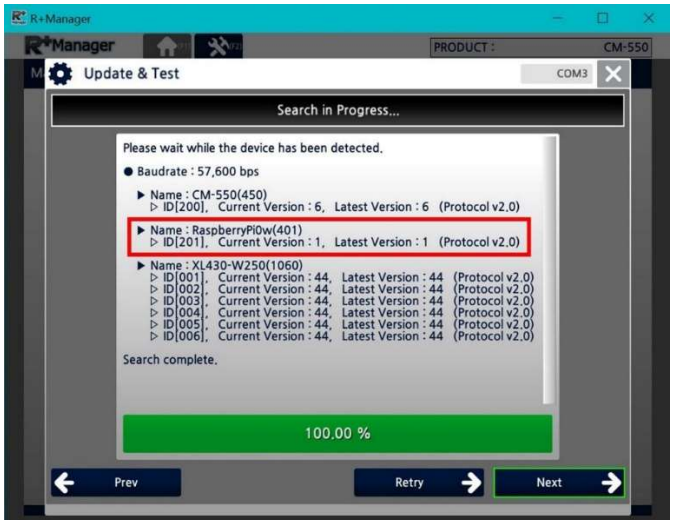


Fig. 1.3 Access to RPi0W via MANAGER Tool.

1.2 2XL430-W250-T and XL430-W250-T

Fig. 1.4 shows a 2XL430-W250-T actuator with mounted frame parts to illustrate its two perpendicular motor axles (http://emanual.robotis.com/docs/en/dxl/x/2xl430-w250/).

Fig. 1.4 Dynamixel 2XL430-W250-T.

It has full 360-degree Position Control including Multi-Turn capability thanks to the use of a magnetic encoder for position sensor (at 12-bit resolution). Of course, it also can be used in Wheel Mode.

Its usage/control is more sophisticated than for the AX-12/18 or XL-320 modules:

- Its Operating Mode (Control Table Address 11) can be set to 4 modes: Velocity Control (i.e. wheel mode), Position Control (i.e. joint mode), Extended Position Control (i.e. multi-turn mode), and PWM Control (for users wanting to control directly the motor's Pulse-Width-Modulation voltages).
- Its Drive Mode (Control Table Address 10) can be set to Normal or Reverse modes, using a Velocity-based or Time-based Profile.
- Additionally, the user can specify its Profile Acceleration (Address 108) and Profile Velocity (Address 112) in combination to obtain 4 types of Position Control modes: STEP,

RECTANGULAR, TRAPEZOIDAL or TIMED (more details in Chapter 2 and YouTube video at https://www.youtube.com/watch?v=4G7DZEVEOmg).

The single-servo version XL430-W250-T can also be used with the ENGINEER kits 1 and 2 and its Control Table information can be found here (http://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#control-table-data-address).

1.3 Software Tools

The CM-550 firmware supports the usual TASK and MOTION tools and now adds a MicroPython engine via a redesigned TASK tool (V. 3.1.1.2 and later) that contains all these 3 sub-tools (see Fig. 1.5). Please note that a downloaded MicroPython program would overwrite any previously downloaded TASK program, and vice versa.

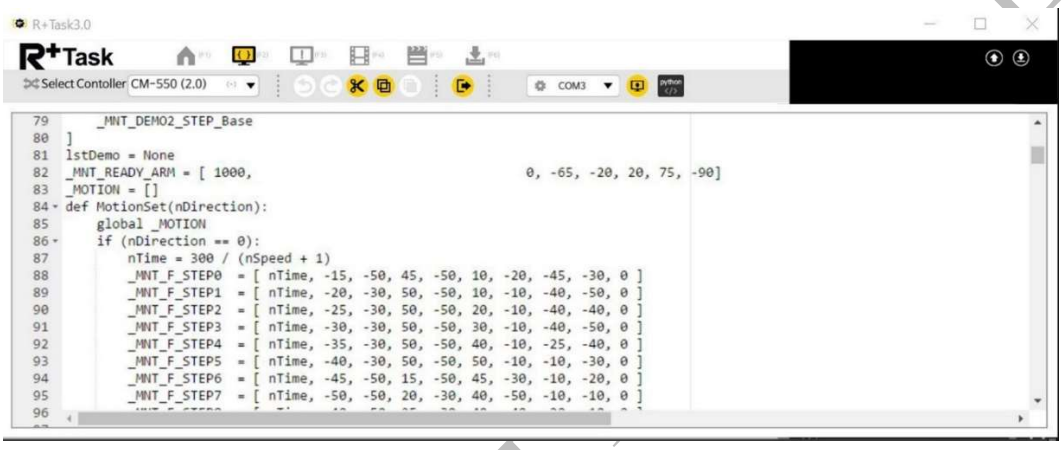


Fig. 1.5 MicroPython programming on the CM-550 via TASK (V. 3.1.1.2 and later).

In the area of Communications Programming, the CM-550 allows the TASK/MicroPython programmer to set up 3 communications ports via TASK/MicroPython's Custom Address commands:

- The TASK MONITOR Port (Control Table Address 35) is where the outputs of PRINT and PRINTLN commands go to (NOTE: in MicroPython, this port is named CONSOLE). The TASK MONITOR Port can be set to BLE (= 0), UART (= 1) or USB (= 2).
- The APP Port (Control Table Address 36) is where communications can be set up between the Remote App and the CM-550. ROBOTIS intended this port to be used with its Mobile App ENGINEER, but the author had found that it works for an application from a Windows PC as well (more application details in Chapter 2). The APP Port can only be set to BLE (= 0) or UART (= 1) and **not to USB currently**. It looks like that ROBOTIS wants to keep the USB Port to be used with the RPi0W that comes with ENGINEER Kit 2.
- The REMOTE Port (Control Table Address 43) is where Remocon packets are received or transmitted by the CM-550 (http://emanual.robotis.com/docs/en/parts/communication/rc-100/#communication-packet). The REMOTE Port can be set to BLE (= 0), UART (= 1) or USB (= 2) (more application details in Chapters 2 and 3).

For the ENGINEER series, ROBOTIS provides many free educational materials but the interested user needs to register his/her CM-550 Serial Number with ROBOTIS at this web link http://en.robotis.com/model/login.php?url=/pdf_project/register.php# before the user can downloading them:

- TASK 3 Programming Outline of 8 lessons (4 pages).
- TASK 3 Programming Curriculum (**T3PC**), with the actual lessons (408 pages).

- Python Online Workbook (**POW**), with 24 lessons and practice questions (282 pages).
- Python Teacher's Guide (**PTG**), with 24 lessons – essentially **POW with answers and extra practice questions** (477 pages).

Building on these ROBOTIS resources, this book series (planned for 2 volumes) is written to help interested users to further utilize the capabilities of the ENGINEER Kits 1 and 2 and even extend these resources whenever possible. This overall goal yields an unusual format for this book series:

- Each **chapter showcases one robot type**, starting in Volume 1 with the "SimpleBot with Arms" (see Fig. 1.6) and progressing towards more sophisticated robots in later chapters and into Volume 2.

Fig. 1.6 Simple Bot with Arms (SBwA).

- Furthermore, within each chapter, the programming tool/environment used also progresses from "simple" like TASK/MOTION and MicroPython on the CM-550 to more "sophisticated" and "enabling" tools such as standard Python and C++ on a Windows PC. In a way, this book is "**configurable**", whereas a user unfamiliar with C++ or Python can just stay with the TASK "path" from one chapter to the next, while a more experienced programmer would choose a C++ or Python "path" instead. Other users may choose or design their "personal" paths depending on their current skill levels and target goals. Another possible scenario is for/when teaching different levels of students while using the **same physical robots** (due to budget or time constraints perhaps), then the instructor can use TASK for beginning students in parallel with Python or C++ for more advanced students.
- The goals of Volume 1 are to establish the foundational robotics concepts and programming techniques for the ENGINEER System using three demonstration robots:
    - The "Simple Bot with Arms" is used to illustrate the basic operations of a purely jointed robot using Dynamixels configured in Position Control.
    - The "Pan-Tilt Commando" is used to illustrate the basic operations of a mixed-control robot that has some Dynamixels configured in Position Control mode and some Dynamixels configured in Velocity Control mode.
    - The "MonoBot" is an unreleased ROBOTIS model that the author used to demonstrate several Dual Robots control approaches.
    - For each robot, multiple projects will be showcased first in TASK codes, then the same projects are re-coded in MicroPython so that readers can appreciate the "translation" requirements and subtleties. Programming features of the CM-550 will be combined with synergistic features from the ENGINEER Mobile App and the RPi0W with Pi Camera. The same projects will also be reviewed and revised by adding the Standard Python and C++ features available at the Desktop PC levels such as the OpenCV library with USB web cam, along with PySerial and Boost.Asio tools and the

ROBOTIS Remocon Packet Protocol to control and acquire sensors data up to two robots simultaneously.

1.4 ROBOTIS Dynamixel Network

ROBOTIS considers all their robotics systems as a connection of Dynamixels (i.e. Smart Servos, with a few Smart Sensors) on a mixed communications network:

- The actuators 2XL430 and XL430 are obvious Dynamixels that can be distinguished from one another via their IDs from "1" to "17" and these IDs can be changed as needed by the user.
- However, there are "special" Dynamixels that have "reserved" IDs and they are the **Hardware Controller CM-550 (ID = 200)**, the **Smart Device (ID = 100)** which can be a Mobile Device (i.e. ENGINEER App), Desktop PC or Single Board Computer. For the ENGINEER System, the **RPi Zero W has a reserved ID of "201"**.
- The CM-550 communicates with the XL430 family of actuators at the 1 Mbps rate but only at 57.6 Kbps with "ID-100" devices and the RPi Zero W (ID = 201).
- This Dynamixel network can handle two types of communication packets: Remocon (http://emanual.robotis.com/docs/en/parts/communication/rc-100/#communication-packet) and Dynamixel Protocol 2 (http://emanual.robotis.com/docs/en/dxl/protocol2/#introduction).

Furthermore, the ROBOTIS Dynamixel Network is hierarchical whereas only 1 device can be the Top-Controller, i.e. the one capable of sending Protocol 2's Instruction Packets for either Read or Write commands to the "other" Dynamixels which can respond with Status Packets as needed. The user currently can set either the CM-550 or the ID-100 Device to be Top-Controller.

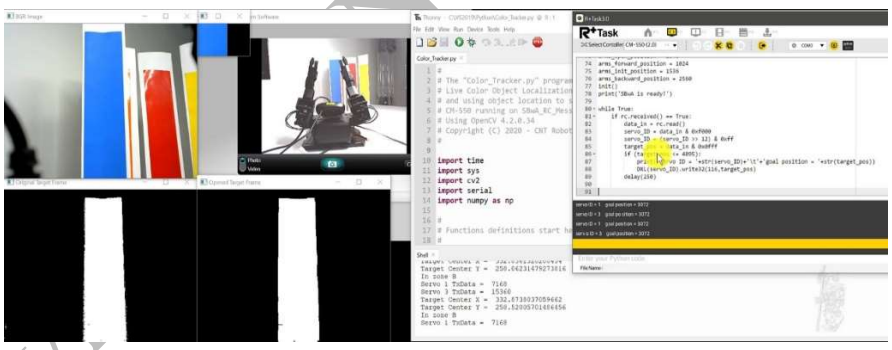


Fig. 1.8 SBwA robot running a Python/MicroPython application using OpenCV and PySerial.

This first-use scenario requires intermediate programming level skills and it is best for applications that need the services of the TASK/MicroPython and MOTION tools and that can accommodate all information transfer between the CM-550 and its Co-Controller within a 16-bit message (unless the programmer uses a multiple Remocon message scheme – see Chapter 3).

This second use scenario would need the integration of the Dynamixel SDK (https://github.com/ROBOTIS-GIT/DynamixelSDK) into the programmer's choice of computer programming language and operating system. This second use scenario is geared towards advanced robotics programmers as all the facilities provided by the TASK/MicroPython and MOTION tools will have to be re-created from function calls into the Dynamixel SDK, but it offers the most flexible and expansive options to the programmer. And who knows, maybe some users have a need to swap between TASK PLAY mode and MANAGE mode for unique robotics applications! ------

Chapter 2: SimpleBot With Arms

Fig. 2.1 shows the original design for the SimpleBot, as it is used in the "Task 3.0 Programming Curriculum" (T3PC) manual from ROBOTIS in the first 5 lessons. SimpleBot's assembly instructions can be found in that document starting on page 367.

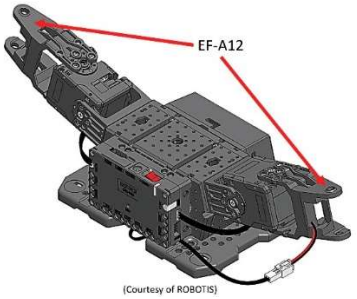


Fig. 2.1 Original Design of SimpleBot.

For a "cooler" look, the author removed Part EF-A12 at two places and replaced them with two "Arms" using the following parts: EF-A14 (2), EF-A05, EF-A06, EF-A07 and EF-A08 (see Fig. 2.2). This robot can be built from parts of the ENGINEER Kit 1.

Fig. 2.2 Details for Left Arm of "SimpleBot with Arms" (SBwA).

The reader is recommended to read or review the T3PC and PTG manuals, before reading on Chapter 2 of this book, as these two manuals illustrate the basic How-To procedures in using R+TASK V. 3.1.x for TASK/MOTION and MicroPython. These topics will not be repeated in this book.

2.1 Using TASK

The T3PC manual illustrates an introductory usage level for "Position Control" of the 2XL430-W250-T actuators via commands such as "Torque On/Off", "Goal Position" and "Present Position". This Section 2.1 presents a more complete treatment of concepts and issues encountered by a "roboteer" when using the 2XL430-W250-T actuator in its Position Control (or Joint) mode. The Velocity Control (or Wheel) mode for this Dynamixel will be described in Chapter 3.

Section 2.1 concentrates on five main parameters which are involved in the programming and subsequent run-time performance of the "2XL430" in Position Control (PC) mode:

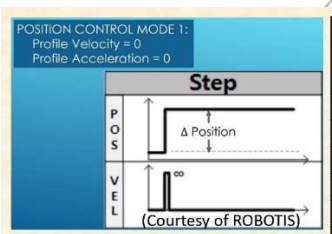
- **Torque Enable** (Address 64 in Control Table for 2XL430).
- **Operating Mode** (Address 11).
- **Drive Mode** (Address 10).
- **Profile Acceleration** (Address 108).
- **Profile Velocity** (Address 112).

2.1.1 "STEP" Position Control mode (PC Mode 1)

For a selected 2XL430, the "STEP" mode is achieved by setting **both** "Profile Acceleration" (PA at Address 108) and "Profile Velocity" (PV at Address 112) to "**0**". To better understand the behind-the-scene process that yielded the run-time performance seen in the previous YouTube video, let's work through an example using **Servo ID=1** initially at rest at **Position 3072**, then its Goal Position (Address 116) is set to **Position 2048** using PC Mode 1. Externally, we would see Servo ID=1 move to Position 2048 at the fastest speed possible among these 4 PC modes.

The author created a TASK program (SBwA_PositionControl_1.tsk3) to perform this example command and to monitor the four parameters VT, PT, PreV and PreP during this servo's movement from Position 3072 to Position 2048. Fig. 2.3 shows the data recorded during the first 100 ms:

- Overall, this TASK code managed to capture data every 10-12 ms cycle while using a BT-410 communication setup between the Windows PC and the CM-550 at 57.6 Kbps.
- VT was found to be set to the same "0" value as ProV, but this "0" value represented a velocity impulse of "infinite" magnitude (only in the mathematical sense of course, in reality the highest electrical current that could be provided by the CM-550 LiPo battery). Therefore, mathematics wise only, the "theoretical" Goal Position was instantaneously achieved, yielding in the "2048" value set for PT.



Time (ms)	Velocity Trajectory	Position Trajectory	Present Velocity	Present Position
5	0	2048	0	3067
15	0	2048	0	3063
25	0	2048	-2	3045
35	0	2048	-13	3021
45	0	2048	-27	2995
55	0	2048	-49	2963
65	0	2048	-72	2931
75	0	2048	-103	2896
86	0	2048	-136	2857
97	0	2048	-164	2817

Fig. 2.3 Typical results for "STEP" Position Control mode during the first 100 milliseconds.

- Interestingly, PreV showed "0" values for the first two readings, although it was clear that the servo was moving from its corresponding PreP data (perhaps these analog velocity voltages were too low to be digitized by the CM-550 A/D converter). Starting from Time=25 ms, PreV showed more and more negative values, meaning that the servo was turning CW and accelerating.
- The PreP parameter also showed that the servo motor was accelerating with increasing larger value differences between consecutive readings.
- Another important point for readers to note that Fig. 2.3 represented "typical" results which would vary slightly from run to run on the actual robot due to uncertainties in communication packet throughputs.

2.1.2 "RECTANGULAR" Position Control mode (PC Mode 2)

For the 2XL430, the "RECTANGULAR" mode is achieved by setting "Profile Acceleration" (PA at Address 108) to "**0**" and "Profile Velocity" (PV at Address 112) to a **positive non-zero value**. See Fig. 2.10 for the values used for various "Profile Velocities" and "Profile Accelerations" in the example program "SBwA_PositionControl_2_mm.tsk3" (the rest of this program uses the same logic as for "SBwA_PositionControl_1_mm.tsk3").

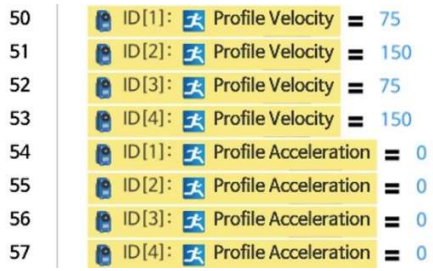


Fig. 2.10 Values used for Profile Velocities and Accelerations in "SBwA_PositionControl_2_mm.tsk3".

Please note that the "shoulder" joints (IDs = 1 & 3) are set to a lower Profile Velocity value than for the "elbow" joints (IDs = 2 & 4), i.e. the "shoulders" will lag behind the "elbows" at run time.

In a similar manner as for Section 2.1.1, let us work through an example using **Servo ID=1** initially at rest at **Position 3072**, then its Goal Position (Address 116) is set to **Position 2048** using PC Mode 2. Externally, we would see Servo ID=1 move to Position 2048 at a slower speed than the one for PC Mode 1.

The TASK program "SBwA_PositionControl_2_mm.tsk3" monitors the four parameters VT, PT, PreV and PreP during this servo's movement from Position 3072 to Position 2048. Fig. 2.11 shows the data recorded during the first 275 ms:

- VT was found to be set to "-75" at Time = 6 ms which was the earliest data point that this program could collect. Ones can see that the embedded controller for Servo 1 computed the "theoretical" values for the pair VT and PT gradually now, but they are always slightly "ahead" of the Present Velocity and Present Position parameters which are essentially the results from the VT+PT settings.

2.1.3 "TRAPEZOIDAL" Position Control mode (PC Mode 3)

The 2XL430's "TRAPEZOIDAL" mode is achieved by setting both "Profile Acceleration" (ProA) and "Profile Velocity" (ProV) to a **positive non-zero value**. See Fig. 2.13 for the values used for various "Profile Velocities" and "Profile Accelerations" in the example program "SBwA_PositionControl_3_mm.tsk3" (the rest of this program also uses the same logic as for "SBwA_PositionControl_1_mm.tsk3" and "SBwA_PositionControl_2_mm.tsk3"). **The goal for using these values is to program the right arm to lag the left arm for all movements.** Please note that ProA's values cannot exceed 50% of ProV's values. This link provides more details about the proper settings for "Profile Velocity" and also for "Profile Acceleration" in a related way (http://emanual.robotis.com/docs/en/dxl/x/2xl430-w250/#profile-velocity112).

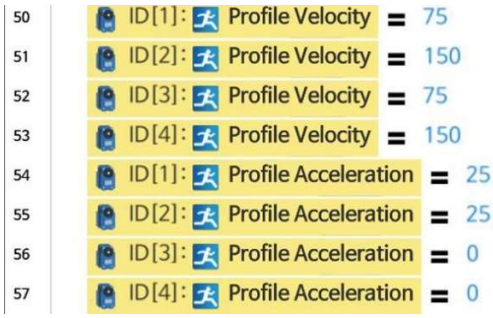
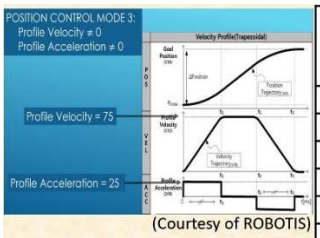


Fig. 2.13 Values used for Profile Velocities and Accelerations in "SBwA_PositionControl_3_mm.tsk3".

In a similar manner as for Section 2.1.2, we can work through an example using **Servo ID=1** initially at rest at **Position 3072**, then its Goal Position (Address 116) is set to **Position 2048** using PC Mode 3. Externally, we would see Servo ID=1 move to Position 2048 at a slower speed than the one for PC Mode 2.

The TASK program "SBwA_PositionControl_3_mm.tsk3" also monitors the four parameters VT, PT, PreV and PreP during this servo's movement from Position 3072 to Position 2048 (using the same logic as shown in the earlier Figs 2.8 and 2.9.



Time (ms)	Velocity Trajectory	Position Trajectory	Present Velocity	Present Position
6	-2	3071	0	3060
57	-22	3062	1	3061
79	-30	3053	0	3060
90	-35	3047	0	3060
100	-39	3040	0	3060
112	-43	3035	0	3058
123	-48	3026	-1	3052
134	-52	3018	-5	3046
145	-56	3009	-9	3040
155	-60	2999	-14	3034
166	-64	2989	-20	3027
177	-69	2977	-27	3018
187	-73	2966	-33	3008
199	-75	2952	-38	2998

Fig. 2.14 Typical results for "TRAPEZOIDAL" Position Control mode 3 during the first 199 ms.

2.1.4 "TIMED" Position Control mode (PC Mode 4)

The last Position Control option is TIME based and this option needs first to be set in the EEPROM Parameter "Drive Mode" (Bit 2 = 1, please review beginning of Section 2.1 if needed). When in a TIME-based Position Control mode, the parameters "Profile Velocity" and "Profile Acceleration" are still used to set up the "Velocity Trajectory" (VT) and "Position Trajectory" (PT), but they will have a completely different meaning: **their numerical values represent "milliseconds" in a TIMED Position Control mode**.

The link http://emanual.robotis.com/docs/en/dxl/x/2xl430-w250/#profile-velocity112 has some information about the TIMED Position Control option, but it is not complete. In actuality, the TIMED PC option can accommodate a RECTANGULAR (PC Mode 4-1) or a TRAPEZOIDAL (PC Mode 4-2) profile in a similar way as for the PC Mode 2 and PC Mode 3 discussed earlier in Sub-Sections 2.1.2 and 2.1.3 respectively.

Let us first start with the RECTANGULAR-TIMED option as shown in Fig. 2.17 which is a section from the TASK code named "SBwA_PositionControl_4-1_mm.tsk3":

- Lines 40-43 show that Bit 2 of Address 10 is set to 1 for each Servo ID 1 to 4, to set them into Time-based profiles. Line 42 additionally set Bit 0 to 1 (i.e. using Reverse mode on ID=3 as before).
- Lines 50-51 set the Profile Velocity for IDs 1 and 2 (i.e. right arm) to a value of 2000 ms, while Lines 52-53 set the Profile Velocity for IDs 3 and 4 (i.e. left arm) to a value of 1000 ms. This means that the right arm will be twice slower than the left arm for all Goal Position moves.
- Lines 54-57 set all Profile Accelerations to "0" to obtain a RECTANGULAR Velocity Trajectory (see Figs. 2.18 and 2.19).

Next, looking back at the previous web link (http://emanual.robotis.com/docs/en/dxl/x/2xl430-w250/#profile-velocity112), ROBOTIS was showing a TRAPEZOIDAL profile (mainly for Velocity-based control) and for Time-based control, they gave only the following formulas:

- t_1 = Profile Acceleration (Address 108).
- $t_2 = t_3 - t_1$
- t_3 = Profile Velocity (Address 112).

Then the author noticed that if t_1 (i.e. Profile Acceleration) was set to "0" (like in the current case), then the TRAPEZOIDAL shape became a RECTANGULAR shape with t_2 as the longer dimension and the previous VTmax as the shorter dimension. However, if this is the case for Velocity-based control, Profile Velocity would become the shorter dimension, and ROBOTIS already provided a formula linking t_2 and Profile Velocity which is:

- $t_2 = 64 * \Delta\text{Position} / \text{Profile Velocity}$, except that the author already determined that "66" needs to be used for his setup.

Integrating all these tidbits of information, the author came up with the following three equations to be used for any **Time-based Profile** whether RECTANGULAR (4-1), TRAPEZOIDAL (4-2) or TRIANGULAR (4-3):

- Equation 1: t_1 = Value at Address 108 (so-called Profile Acceleration)
- Equation 2: t_3 = Value at Address 112 (so-called Profile Velocity)
- Equation 3: $66 * \Delta\text{Position} = \text{VTmax} * t_1 + \text{VTmax} * (t_3 - 2 * t_1)$

  Eq. 3 just represents the "area" under a trapezoidal profile as defined in Fig. 2.20 where the term [VTmax * t_1] accounts for the areas of the 2 triangular "wings" of the trapezoid and the term [VTmax * (t_3 – 2 * t_1)] represents the rectangular middle part of the trapezoid.

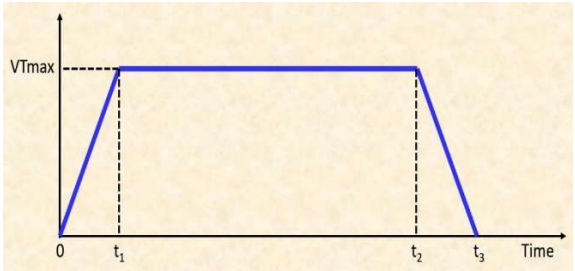


Fig. 2.20 Velocity Trajectory definition for a Time-Based Position Control option (i.e. PC Modes 4-1, 4-2 or 4-3).

2.1.5 Position Control with SyncWrite

"SyncWrite" is a feature of the Dynamixel Protocol 2 (http://emanual.robotis.com/docs/en/dxl/protocol2/#sync-write) which had been available to users of the ROBOTIS Dynamixel SDK for many years, but this is the first time that this feature is implemented at the TASK and MicroPython levels. It is available for the CM-550 only at present (http://emanual.robotis.com/docs/en/software/rplustask3/task_parameters/#syncwrite). "SyncWrite" allows the programmer to control multiple Dynamixels with a single (and long) Instruction Packet which is very helpful, for example, in the synchronization of Dynamixels comprising a robotic arm or each side of a multi-wheeled vehicle (see Chapter 3).

The TASK program "SBwA_PositionControl_SyncWrite.tsk3" uses a "generic" Function named "SW_Goal_Position" to control all 4 servos of the SBwA robot simultaneously (see left panel in Fig. 2.23). In this case, the same type of command (i.e. Goal Position at Address 116 – Line 91) was needed for all 4 servos and the right panel in Fig. 2.23 showed how to use Function "SW_Goal_Position" in the Main part of this TASK program. At present, TASK does not support indexed arrays so individual parameters such as "Servo1", "Servo2", "Servo3" and "Servo4" need to be used (Lines 12-15). When using MicroPython (see Section 2.2) this function can be written more compactly using a FOR LOOP operating on an indexed array of Goal Positions.

2.1.6 Motion Play Features: Motion Speed, Stop Page, Joint Offset

In the author's opinion, "MOTION Programming" is the most prominent feature in the RoboPlus software suite since Version 1 was available circa 2005 for the BIOLOID series. It is indispensable when ones deal with a jointed robot such as a humanoid one, and the use to the UNITY engine since MOTION V.2 has made the creation of Motion Units/Pages/Groups so much easier and efficient for the end-user. The author recommends readers to read or review Lessons 6 through 8 of the TASK 3 Programming Curriculum (T3PC) before proceeding further in this Section 2.1.6. For a more thorough illustration of the MOTION tool with the MINI system, please access the author's work on the MINI system (Thai, 2020). The information described in this work for the MINI is still applicable to the ENGINEER kits except for the following additions and changes:

- The SPEED of MOTION PLAY can be adjusted during run-time for the CM-550.
- MOTION V.3 adds two new options -2 and -3 for MOTION STOP PAGES for the CM-550.
- The author's favorite bug/feature named JOINT OFFSET gets drastically changed for its use on the XL430 family of actuators and the CM-550.

Thus, the emphases in this Section 2.1.6 will be on the above topics.

The file "SBwA_MotionPlay.mtn3" contains the Motions defined for the "SBwA" robot:

- Four Single-Keyframe Motion Units (MU) were defined: "Init_Pose", "Pose_1", "Pose_2", and "Pose_3" and they corresponded to the Poses with the same name used in various TASK programs of Sections 2.1.1 through 2.1.5.
- Six Motion Lists/Pages were created based on the previous 4 Motions Units. The first four Motion Pages were the same as the four Motion Units. The 4th Motion Page is named "Pose_1-3" and it is a sequential list of the Motion Units "Pose_1", "Pose_2" and "Pose_3". The Motion Unit "Init_Pose" is used as **Exit Motion Unit** for each of these MUs. The 5th Motion Page is named "Repeat_Pose_1-3" and it is an Endless Loop of the MUs "Pose_1", "Pose_2" and "Pose_3". "Init_Pose" is also used as Exit MU for "Repeat_Pose_1-3".
- The Motion Group used for TASK programs in this Section 2.1.6 is named "SBwA_Project1".

2.1.7 Using Smart Device

As previously mentioned in Section 1.4, the CM-550 can be programmed to interact with a Mobile Device (Android and iOS) via the App named ENGINEER (https://apps.apple.com/us/app/r-engineer/id1475713920 or https://play.google.com/store/apps/details?id=com.robotis.robotisEngineer) by integrating "Smart Device" commands into standard TASK codes.

There is an extensive list of "SMART DEVICE" commands as shown in this web page (http://emanual.robotis.com/docs/en/software/rplustask3/task_parameters/#smart-device) and of example TASK codes showing how to use them. However, currently (June 2020) this information is not yet updated for **recent** Smart Device updates (**for CM-550 only**) from TASK 3.1 (some of these CM-550 specific commands will be showcased in Chapter 3).

As an example project, the program "SBwA_SD_CameraTracker.tsk3" illustrates how to create TASK codes to make the front camera of a Mobile Device track an object of User-assigned Color and to move the arms of the SBwA robot accordingly to the data sent from the same Mobile Device. This program also shows two ways to display texts on the Mobile Device's display.

As far as TASK 3.1 is concerned, there are two options to access the Mobile Device Display Screen: Low-Res (for CM-550 and older controllers such as CM-50/150/530/904) and Hi-Res (CM-550 only). The Hi-Res option will be described in Chapter 3 and in this Chapter 2 only the Low-Res option is used whereas the Mobile Device's Display Screen is divided into 25 zones arranged in a 5x5 grid in either Portrait or Landscape mode (see Fig. 2.27).

2.2 Using MicroPython

As previously mentioned in Section 1.3, ROBOTIS has released several educational materials for using MicroPython with the ENGINEER Kits 1 and 2: the POW and PTG manuals. **The author needs the reader to review these materials before reading the rest of Section 2.2**.

A MicroPython program will require about the same number of statements (or less – depending on the programmer's Python skills) than a TASK program to get the CM-550 to do a typical action, and its execution speed seems to be on par with an equivalent compiled TASK code (i.e. compiled C/C++ code), so the author is quite impressed with the Embedded MicroPython Engine inside the CM-550. When combined with Standard Python on the PC or SBC side (see Section 2.3), it will allow the user to tap into the vast resources of other Python packages available at the Python Software Foundation (https://pypi.org/ and https://docs.python.org/3/contents.html). The web site Python Central is also a great resource for learning and practicing Python programming (https://www.pythoncentral.io/).

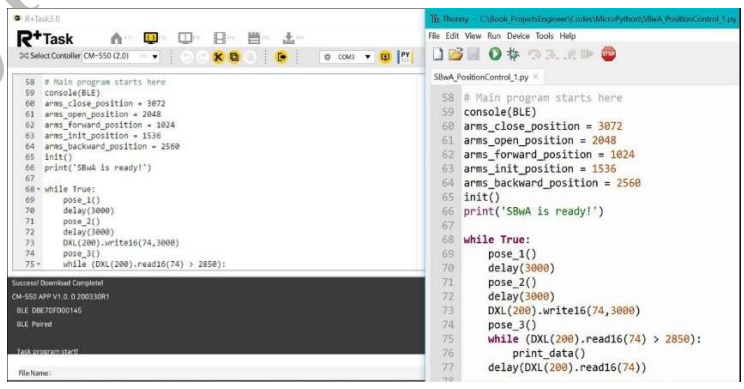


Fig. 2.35 Author's preferred solution for MicroPython Development.

The CM-550's MicroPython editor is based on PyPlay and it is quite minimal as far as the User Interface is concerned. Thus, the author prefers Thonny (right panel in Fig. 2.35) to do the program editing and TASK in MicroPython mode to compile/run his example codes (left panel in Fig. 2.35).

2.2.1 "STEP" Position Control mode (PC Mode 1)

For a selected 2XL430, the "STEP" mode is achieved by setting **both** "Profile Acceleration" (PA at Address 108) and "Profile Velocity" (PV at Address 112) to "**0**". To better understand the behind-the-scene process that yielded the run-time performance seen in the previous YouTube video, let's work through an example using **Servo ID=1** initially at rest at **Position 3072**, then its Goal Position (Address 116) is set to **Position 2048** using PC Mode 1. Externally, we would see Servo ID=1 move to Position 2048 at the fastest speed possible among these 4 PC modes.

The author created a MicroPython program "SBwA_PositionControl_1.py" to perform this example command and to monitor the four parameters **vel_traj**, **pos_traj**, **pre_vel** and **pre_pos** during this servo's movement from Position 3072 to Position 2048.

A reminder for readers to visit http://www.cntrobotics.com/engineer for access options to the source codes, and let's have a closer look at the program "SBwA_PositionControl_1.py".

Fig. 2.36 illustrates the code segments for importing the "pycm" module (Line 6) and the steps needed in the Main Section of "SBwA_PositionControl_1.py":

- Line 59 specifies that CONSOLE outputs are directed to the BLE device/port.
- Lines 60-64 specify the CONSTANTS used in this program.
- Line 65 calls the User Function **init()** (see Fig. 2.37).
- Line 66 prints out an informational message for the programmer.

```
6  from pycm import *

58 # Main program starts here
59 console(BLE)
60 arms_close_position = 3072
61 arms_open_position = 2048
62 arms_forward_position = 1024
63 arms_init_position = 1536
64 arms_backward_position = 2560
65 init()
66 print('SBwA is ready!')
67
68 while True:
69     pose_1()
70     delay(3000)
71     pose_2()
72     delay(3000)
73     DXL(200).write16(74,3000)
74     pose_3()
75     while (DXL(200).read16(74) > 2850):
76         print_data()
77     delay(DXL(200).read16(74))
```

Fig. 2.36 Main Section of "SBwA_PositionControl_1.py".

- Lines 68-77 list the steps used for the Main Algorithm (e.g. Main Endless Loop):

- First, Function **pose_1()** (see Fig. 2.38) is called to set the SBwA robot into its Pose 1 (Line 69).
- Line 70 delays the CM-550 for 3000 ms.
- Line 71 calls Function **pose_2()** (see Fig. 2.38) and Line 72 delays for another 3000 ms.

In conjunction with Stop Pages which "stop" the complete robot when executed, there is an "advanced" Dynamixel parameter (*not listed in the Control Tables for XL430 and 2XL430 actuators*) that allow the programmer to **deactivate specific actuators** from the motions previously defined for them in the current Motion Group used by the programmer. Currently (Fall 2020), there is no official ROBOTIS name for this parameter and users would have to use Custom Write commands into specific Control Table RAM addresses defined by the following formula:

- RAM **Address = 1016 + ID**. For example, if ID = 2, the Address 1016 + 2 = 1018 should be used to activate/deactivate Servo 2.
- A Byte value should be used with this Custom Write command. By default, a value of "1" is written there, so upon robot power up all Servos are enabled. During run-time, if a value of "0" is set to a specific Servo, that specific Servo would just "hold" the Position where it was at when it received this command for all subsequent Motion Pages called to be played.
- For example, a Custom Write command such as ADDR[1018(b)] = 0 can be used to deactivate Servo 2.

2.2.7 Using Smart Device

As previously mentioned in Section 1.4, the CM-550 can be programmed to interact with a Mobile Device (Android and iOS) via the App named ENGINEER (https://apps.apple.com/us/app/r-engineer/id1475713920 or https://play.google.com/store/apps/details?id=com.robotis.robotisEngineer) by integrating "Smart Device" commands into standard MicroPython codes.

When using TASK (Section 2.1.7), the reader can access an extensive list of "SMART DEVICE" commands and example TASK codes showing how to use them at this web page (http://emanual.robotis.com/docs/en/software/rplustask3/task_parameters/#smart-device, but so far (June 2020) no such documentation exists for the MicroPython tool, **except for the MicroPython example codes provided for the ENGINEER Kit 2, which were provided without any in-line comments!**. So at present, the English reader will have to rely on the R+SMART Control Table (RSCT) which unfortunately is only in Korean (http://support.robotis.com/ko/software/mobile_app/r+smart/smanrt_manual.htm#Actuator_Address_0B3) but it can be easily translated into English when using the Google Chrome web browser. The reader will later see that "Smart Device" commands in MicroPython will be mostly "Direct Address" Read/Write functions relying on the Addresses and Sizes of Smart Parameters listed in this R+SMART Control Table.

```
while True:
    linePosition = smart.read8(10110)  # obtain latest linePosition from "LineDetectionArea" function

    while (linePosition != 0):
        nPosition = 18    # Position [3,4]
        nItem = 0   # i.e. Item Null
        nSize = 50
        nColor = 2  # Black Color
        nText = nPosition + nItem * 256 + nSize * 65536 + nColor * 16777216
        smart.display.text(nText)
        display_color_position(lineColor, linePosition)
        if (linePosition == 3):
            sw_goal_pos(arms_up_position, arms_init_position, arms_up_position, arms_init_position)
        elif (linePosition == 1):
            sw_goal_pos(arms_mid_1_position, arms_init_position, arms_down_position, arms_init_position)
        elif (linePosition == 2):
            sw_goal_pos(arms_mid_2_position, arms_init_position, arms_down_position, arms_init_position)
        elif (linePosition == 4):
            sw_goal_pos(arms_down_position, arms_init_position, arms_mid_2_position, arms_init_position)
        elif (linePosition == 5):
            sw_goal_pos(arms_down_position, arms_init_position, arms_mid_1_position, arms_init_position)
        delay(500)
        linePosition = smart.read8(10110)   # obtain latest linePosition from "LineDetectionArea" function

    nPosition = 18    # Position [3,4]
    nItem = 2   # i.e. "No Color Found"
    nSize = 50
    nColor = 1  # White Color
    nText = nPosition + nItem * 256 + nSize * 65536 + nColor * 16777216
    smart.display.text(nText)
    sw_goal_pos(arms_down_position, arms_init_position, arms_down_position, arms_init_position)
```

Fig. 2.65 "Essential" Algorithm used in "SBwA_SD_CameraTracker.py".

2.3 Using OpenCV-Python & PySerial on Desktop

So far in Sections 2.1 and 2.2, whenever we used the Virtual Remote Controller RC-100, we used it to send status information about the 10 RC Buttons U-D-L-R-1-2-3-4-5-6, i.e. a 10-bit message, from a Desktop PC to a TASK or MicroPython program running on the CM-550. These **Remocon** packets are actually 6-bytes long following a protocol described at this web link (http://emanual.robotis.com/docs/en/parts/communication/rc-100/#communication-packet) which shows that the "useful" message is really only **16-bit** long.

This protocol (also known as Zigbee SDK) had been created by ROBOTIS a long time ago for its BIOLOID EXPERT Kit (c. 2005) which was based on the CM-5 Controller (i.e. "ancient" technology!). However, this protocol is still quite useful as it allows to bring in Standard Python (therefore a multitude of application areas) from a Desktop PC or Single Board Computer (SBC) to work together with the ROBOPLUS Software Suite (TASK/MOTION/ENGINEER App). As previously hinted in Section 1.4, the following Dynamixel Network Configuration is implemented for Section 2.3:

2.3.1 Python Implementation of Remocon Packet Creation on PC Side

Currently, the SDK for using Remocon packets is called the Zigbee SDK V. 1.0.2 (c. 2010) and usage information can be found at this web link https://github.com/ROBOTIS-GIT/emanual/blob/master/docs/en/software/embedded_sdk/zigbee_sdk.md. Unfortunately, it is also called the ZIG2Serial SDK (https://github.com/ROBOTIS-GIT/emanual/blob/master/docs/en/software/embedded_sdk/zigbee_sdk.md#zig2serial) because during the ROBOTIS early years (c. 2005), Zigbee technology was used as wireless technology for its BIOLOID series with the Controllers CM-5, CM-510 and CM-530, and communications hardware such as ZIG-100/110A, ZIG2SERIAL and USB2DYNAMIXEL were used during that period. Furthermore, the previous GITHUB web site provides information for C/C++/C# implementations but there is no Python version for this SDK at present. After some research, the author found that basic functionalities for the ROBOTIS Remocon Packet can be reconstructed using the Python BYTEARRAY class (https://docs.python.org/3.7/library/stdtypes.html?highlight=bytearray#bytearray) and that the PYSERIAL module (https://pypi.org/project/pyserial/, https://pyserial.readthedocs.io/en/latest/pyserial.html) can be used to send them from the Desktop PC to the CM-550 using BT-210 or BT-410 receivers which are the current Bluetooth modules used for the ENGINEER kits.

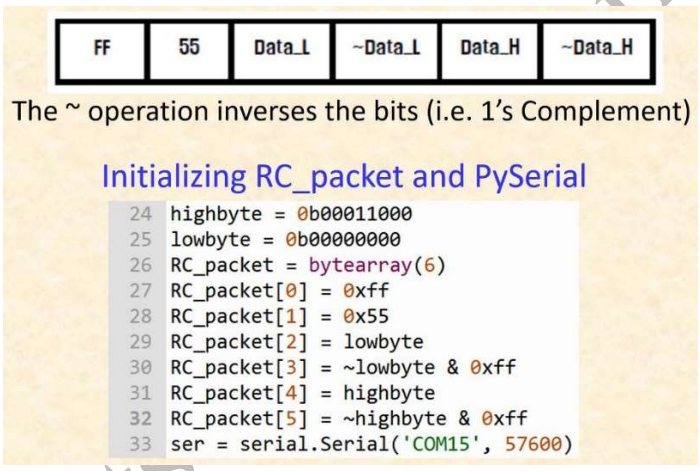


Fig. 2.66 Python's BYTEARRAY implementation of ROBOTIS Remocon Packet.

The bottom part of Fig. 2.66 shows the Python's implementation of the ROBOTIS Remocon packet using the **bytearray** class.

2.3.2 Handling of Remocon Packet on CM-550 Side

When the CM-550 receives a Remocon Packet "shaped" by the procedures previously described in Section 2.3.1, it goes through a decoding/unpacking process which is slightly different in TASK vs. MicroPython because TASK does not support any bit-shift operator. Thus, the TASK programmer needs to fall back on the arithmetic operators like Multiplication for Bitwise Left-Shift (<<) and Division for Bitwise Right-Shift (>>).

Fig. 2.68 shows how a typical Remocon Packet is handled in TASK, with GP-type of packets described in the top part of Fig. 2.68 and XY-type of packets described in its bottom part:

- Both procedures first use a WAIT WHILE Loop based on the "Remocon Data Arrived" Flag, i.e. waiting for it to turn TRUE when a new Remocon Packet has just been received by the CM-550.

- If it is a GP-type packet (top area of Fig. 2.68), Line 24 extracts Bits 12-15 from Parameter "MessageReceived" (counting from Bit 0 being the right-most bit) using the Bitwise **&** Operator and the Constant "1111 0000 0000" (or decimal 61440). Next, Line 25 right-shifts this temporary result by 12 bits with the use of the Division Operator and the Constant "1 0000 0000 0000" (or decimal 4096) and saved this final result into Parameter "ServoID". Then with Line 26, Parameter "Goal Position" is next extracted from "Message Received" using the Bitwise **&** Operator and the Constant "1111 1111 1111" (or decimal 4095).

SBwA_RC_MS_GP_SD.tsk3

```
21  WAIT WHILE ( Remocon Data Arrived == FALSE (0) )
22  MessageReceived = Remocon RXD
23  // Decode Message Received
24  ServoID = MessageReceived & 0000 0000 0000 0000 1111 0000 0000 0000
25  ServoID = ServoID / 0000 0000 0000 0000 0001 0000 0000 0000
26  TargetPosition = MessageReceived & 0000 0000 0000 0000 0000 1111 1111 1111
```

SBwA_RC_MS_XY_SD.tsk3

```
21  WAIT WHILE ( Remocon Data Arrived == FALSE (0) )
22  MessageReceived = Remocon RXD
23  // Decode Message Received
24  XY_Key = MessageReceived & 0000 0000 0000 0000 1111 0000 0000 0000
25  XY_Key = XY_Key / 0000 0000 0000 0000 0001 0000 0000 0000
26  // Only Target_X is needed in this project
27  IF ( XY_Key == 1 )
28  {
29    Target_X = MessageReceived & 0000 0000 0000 0000 0000 1111 1111 1111
```

Fig. 2.68 Decoding/Unpacking of Remocon Packet on CM-550's side using TASK codes.

The author used a Logitech C270 webcam mounted between the arms of the SBwA robot to search for a user-defined colored object. Fig. 2.70 displays a typical run-time snapshot of the setup used for the OpenCV projects.

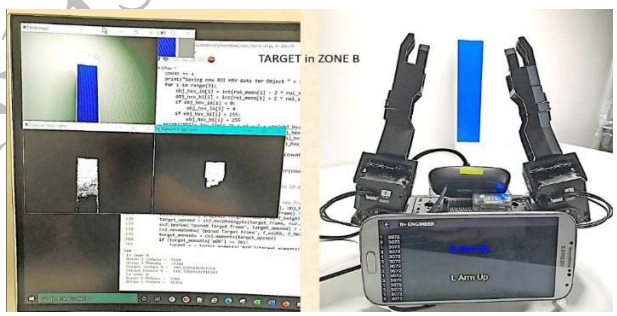


Fig. 2.70 Run-time setup for OpenCV projects in Section 2.3.

The OpenCV sections are the same in the example programs **SBwA_Color_Tracker_GP.py** and **SBwA_Color_Tracker_XY.py**, thus they are described only once herein.

Fig. 2.71 illustrates needed initialization and checking steps to use the webcam as a VideoCapture object:

- **Section 1:**

- Lines 54 and 55 initialize two 1-D arrays, with 3 elements each, named **obj_hsv_lo[3]** and **obj_hsv_hi[3]** which will be used to store the tracked object's HSV Lower and Higher Limit values respectively, in a later code section.
- Lines 56-57 initialize Parameters **scan_mode** and **count** to zero.
- Line 59 creates a 2-D array named **kernel** with a **20x20** size and fills it up with **1** as unsigned 8-bit integers, using the NumPy method named **np.ones()**.

OpenCV Sections in SBwA_Color_Tracker_GP.py & SBwA_Color_Tracker_XY.py

(1)

```
52 # OpenCV section
53 # Pixel values limits to be set for object to be tracked
54 obj_hsv_lo = [None] * 3
55 obj_hsv_hi = [None] * 3
56 scan_mode = 0
57 count = 0
58 # Kernel size used for image processing (20x20 pixels)
59 kernel = np.ones((20,20),np.uint8)
```

(2)

```
82 vid_cam = cv2.VideoCapture(0)
83 if(not vid_cam.isOpened()):
84     print("Error connecting to camera - Exiting program")
85     vid_cam.release()
86     sys.exit()   # terminate program
87
88 got_frame, vid_frame = vid_cam.read()  # picking up a new video frame
89 if not got_frame:   # error capturing video frame
90     print("No video frame captured - Exiting program")
91     sys.exit()  # terminate program
```

Fig. 2.71 OpenCV Sections 1 and 2 used in "SBwA_Color_Tracker_GP.py" and "SBwA_Color_Tracker_XY.py".

2.4.1 Using ZigBee SDK with VS 2019

One benefit with using the ZigBee SDK with C/C++ is that its API is well developed with many utility functions (https://emanual.robotis.com/docs/en/software/embedded_sdk/zigbee_sdk/).

This SDK is composed of a Hardware Abstraction Layer (HAL) defined by the files "zgb_hal.h" and "zgb_hal.c". HAL works at the Windows OS level to do the opening and closing of COM ports using a DCB structure (https://docs.microsoft.com/en-us/windows/win32/devio/configuring-a-communications-resource, https://docs.microsoft.com/en-us/windows/win32/api/winbase/ns-winbase-dcb). HAL also takes care of the actual Transmission and Receiving of the Remocon Packet via Methods **zgb_hal_tx()** and **zgb_hal_rx()** (see Fig. 2.83). Fortunately for applications programming, it is not necessary to work at this level!

For applications programming with this SDK, we do however need to get familiar with the contents of the files "zigbee.h" and "zigbee.c". Fig. 2.84 lists the contents of the file "zigbee.h":

- There are 2 Device Control Methods **zgb_initialize()** and **zgb_terminate()**.
- There are 3 Communications Methods **zgb_tx_data()**, **zgb_rx_check()** and **zgb_rx_data()**.
- The Constants for the RC-100 Buttons U-D-L-R-1-2-3-4-5-6 are also defined in this file (Lines 19-28).

```
#ifndef _ZIGBEE_HAL_HEADER
#define _ZIGBEE_HAL_HEADER

#ifdef __cplusplus
extern "C" {
#endif

int zgb_hal_open( int devIndex, float baudrate );
void zgb_hal_close();
int zgb_hal_tx( unsigned char *pPacket, int numPacket );
int zgb_hal_rx( unsigned char *pPacket, int numPacket );

#ifdef __cplusplus
}
#endif

#endif
```

```
int zgb_hal_open( int devIndex, float baudrate )
{
    // Opening device
    // devIndex: Device index
    // baudrate: Real baudrate (ex> 115200, 57600, 38400...)
    // Return: 0(Failed), 1(Succeed)

    DCB Dcb;
    COMMTIMEOUTS Timeouts;
    DWORD dwError;
    char PortName[15];
```

Fig. 2.83 Hardware Abstraction Layer for ROBOTIS ZigBee SDK.

```
#ifndef _ZIGBEE_HEADER
#define _ZIGBEE_HEADER


#ifdef __cplusplus
extern "C" {
#endif

////////////// device control methods /////////
int __stdcall zgb_initialize( int devIndex );
void __stdcall zgb_terminate();

////////// communication methods ////////////
int __stdcall zgb_tx_data(int data);
int __stdcall zgb_rx_check();
int __stdcall zgb_rx_data();

////////// define RC-100 button key value ////
#define RC100_BTN_U     (1)
#define RC100_BTN_D     (2)
#define RC100_BTN_L     (4)
#define RC100_BTN_R     (8)
#define RC100_BTN_1     (16)
#define RC100_BTN_2     (32)
#define RC100_BTN_3     (64)
#define RC100_BTN_4     (128)
#define RC100_BTN_5     (256)
#define RC100_BTN_6     (512)

#ifdef __cplusplus
}
#endif
```

Fig. 2.84 Header file "zigbee.h" for ROBOTIS ZigBee SDK.

2.4.2 Project "SBwA_Color_Tracker_GP.cpp"

Let us now get into the details of integrating OpenCV and ZigBee SDK into the project/program "SBwA_Color_Tracker_GP.cpp" which has a similar functionality as "SBwA_Color_Tracker_GP.py" from Section 2.3.4.

Fig. 2.86 lists all the needed C++ Preprocessor Directives, in particular:

- Line 12 enables Function Calls to the ZigBee API (https://emanual.robotis.com/docs/en/software/embedded_sdk/zigbee_sdk/).
- Lines 19-22 enables Function Calls to the OpenCV 4.2 API (https://docs.opencv.org/4.2.0/).
- Line 24 defines COM**15** to be used for the author's BT-210 receiver which is connected to the UART Port on the CM-550. The reader would need to revise this COM number to fit the reader's usage conditions.

- Line 25 defines the TimeOut period (= 1000 ms) for the program to wait for a connection to the BT-210 to be made. The author had found that sometimes more than 1000 ms were needed for his setup.

```
 9  #include <windows.h>
10  #include <stdio.h>
11  #include <conio.h>
12  #include "zigbee.h"
13  #include <iostream>
14  #include <sstream>
15  #include <string>
16  #include <fstream>
17  #include <cstdlib>
18  #include <iomanip>
19  #include "opencv2/core/core.hpp"
20  #include "opencv2/highgui/highgui.hpp"
21  #include "opencv2/imgproc/imgproc.hpp"
22  #include "opencv2/core/types.hpp"
23
24  #define DEFAULT_PORTNUM     15  // COM3
25  #define TIMEOUT_TIME        1000 // msec
26
27  using namespace std;
28  using namespace cv;
```

Fig. 2.86 Various Preprocessor Directives used in "SBwA_Color_Tracker_GP.cpp".

Fig. 2.96 lists OpenCV procedures used in Part C of the Main Endless Loop to isolate the previously defined Color Blob and to compute its Area Mass Center's screen coordinates **target_x** and **target_y**:

- Line 246 makes sure this code section is executed only when **scan_mode** is set to **1**.
- Line 249 picks up a fresh color image frame **src** from the video camera **vid_cam**.
- Line 251 converts **src** (BGR colors) to **hsv** (HSV colors) and Line 252 "splits" **hsv** to its separate "color-channel" (i.e. monochrome) images contained a special structure **HSVchannels** defined back in Line 138 of Fig. 2.89 (https://docs.opencv.org/4.2.0/d2/de8/group__core__array.html#ga8027f9deee1e42716be8039e5863fbd9).
- The main reason for the author to use the HSV Color Space is that only the Hue image **HSVchannel[0]** would be needed for Color Tracking in this project. OpenCV Function **inRange()** is used for this task as shown in Line 255 (https://docs.opencv.org/4.2.0/d2/de8/group__core__array.html#ga48af0ab51e36436c5d043400e036ce981). Function **inRange()** takes in 3 inputs: **HSVchannel[0]**, **obj_hsv_lo[0]** and **obj_hsv_hi[0]** and outputs a new binary image named **target** which is displayed by Line 256 using **imShow()**. However, **target** usually contains other "noise" pixels and/or "smaller" unwanted objects (see Fig. 2.70 and lower left B&W window frame) and thus needs another image processing step to "clean" it up.
- This "clean-up" task is performed by OpenCV Function **morphologyEx()** at Line 257 using as inputs: image frame **target**, option's parameter **MORPH_OPEN** and the special 2D array **kernel** defined back in Line 37 of Fig. 2.87. This function outputs a new binary image named **target_opened** which is "cleaned-up" but it also "losts" quite a few pixels due to this operation (see Fig. 2.70 – lower right B&W window frame). Fortunately, in this project, the critical parameter which is the screen width/x-coordinate of the target's mass center can still be used effectively by the robot to track the color target with its arms.

```
246     if (scan_mode == 1)
247     {
248         // OPENCV SECTION
249         vid_cam >> src; //read current video frame
250         // Convert src (BGR) to hsv
251         cvtColor(src, hsv, COLOR_BGR2HSV);
252         split(hsv, HSVchannels); // Splits the image into 3 channel images
253
254         // Thresholding for chosen object in Hue channel only
255         inRange(HSVchannels[0], obj_hsv_lo[0], obj_hsv_hi[0], target);
256         imshow("Original Target Frame", target);  // display original target frame
257         morphologyEx(target, target_opened, MORPH_OPEN, kernel);
258         imshow("Opened Target Frame", target_opened); // display opened target frame
259         target_moments = moments(target_opened);
260         if (target_moments.m00 >= 30)
261         {
262             target_x = target_moments.m10 / target_moments.m00;
263             target_y = target_moments.m01 / target_moments.m00;
264             cout << "Target Center X = " << target_x << endl;
265             cout << "Target Center Y = " << target_y << endl;
266         }
```

Fig. 2.96 Part C (OpenCV) of Main Endless Loop in "SBwA_Color_Tracker_GP.cpp".

- Next, this B&W image **target_opened** is used as input to another OpenCV Function named **moments()** to compute various Area/Spatial Moment parameters which are recorded in the structure **target_moments** (Line 259) – see more details at link https://docs.opencv.org/4.2.0/d8/d23/classcv_1_1Moments.html#details.
- If the member parameter **target_moments.m00** is found to be larger than 30 pixels (Line 260), i.e. a "valid" color target had been found by OpenCV, then the IF structure defined by Lines 260-266 is activated to compute and print out Parameters **target_x** and **target_y**.

2.4.3 Project "SBwA_Color_Tracker_XY.cpp"

If the user prefers to send from the PC the Target Screen Coordinates **target_x** and **target_y** instead, then the user can load up the program "SBwA_Color_Tracker_XY.cpp". All the OpenCV and ZigBee SDK procedures will be the same as illustrated in Section 2.4.2, the only difference is in how the Remocon Packet is prepared and sent away to the CM-550.

Fig. 2.98 shows the relevant details for Part D of this "XY" project:

- A new Flag Parameter **xy_key** is needed and set to 0 at the beginning of the program "SBwA_Color_Tracker_XY.cpp" (Line 134 in the source code).
- When all the OpenCV procedures are successful in computing a valid value for **target_x**, the code segment (Lines 271-287) is activated:
    - Parameter **xy_key** is set to 1 (Line 274) and then this value is shifted 12 bits to the left in Parameter **TxData** (first expression in the RHS of Line 275).
    - Next, the current value of **target_x** is inserted into the lower 12 bits of **TxData**.
    - **TxData** is then sent via SDK Function **zgb_tx_data()** and checked for transmission failure by Lines 277-278.
    - A short delay on 100 ms (Line 279) is then used so as not to overwhelm the Bluetooth COM service.
    - Lines 282-286 repeat a similar procedure to send **target_y** data (*This code section can also be commented out by the user as **target_y** is not used by the CM-550 in this project*).

Chapter 3: Commando with Pan-Tilt Platform + Camera

Fig. 3.1 shows the author's mechanical modifications to the original design for the Commando robot which came with the ENGINEER Kit 2:

1. Frame part EF25-F18 replaces EF25-F23 used in the original Commando robot (Step 12 in the Assembly Manual for Commando) and works as a Pan-Tilt frame for the Pi Camera.
2. An additional XL430-W250-T is used as Servo 6 which acts as a Pan-Servo while the original Servo 5 acts as a Tilt-Servo.
3. Wheels, i.e. rubber tires, are used instead of the caterpillar tracks.

A BT-210 receiver (not provided in the ENGINEER Kits 1 and 2) is connected to the UART Port of the CM-550 and allows serial communications to the Desktop PC, while the built-in BT-410 is used to communicate with a Mobile Device via the ENGINEER App. The RPi Zero W SBC communicates to the CM-550 via the OTG USB cable as provided by ROBOTIS, and also to the Mobile Device via WiFi (https://emanual.robotis.com/docs/en/edu/engineer/kit2_reference/#setting-video-streaming-on-robotis-engineer-app) for video streaming.

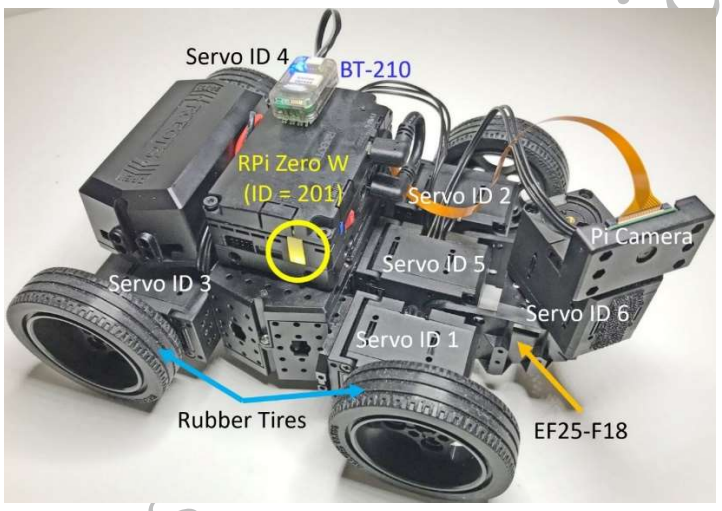


Fig. 3.1 Pan-Tilt Commando (PTC) robot with RPi0W and Pi Camera.

The use of the RPi Zero W will **require more care** from the user as he/she will have to **wait up to 4 minutes** upon powering up the Commando robot as configured in Fig. 3.1, because the RPi SBC does take a long time to boot up and to execute its default ROBOTIS application software. The Commando will be ready for further use when the **START/USER LED on the CM-550 lights up YELLOW** as shown in Fig. 3.1 (inside the Yellow Circle). Then the user can use the robot to work with the ROBOTIS tools such as MANAGER and TASK as normal. Fig. 3.2 shows that MANAGER sees the RPi-Zero-W as another **DYNAMIXEL with an ID=201** (which should not be changed by the user).

Please note that if the user runs MANAGER before the CM-550's START/USER LED turns YELLOW, the MANAGER tool will only show the connections to the CM-550 and the XL430-W250 servos.

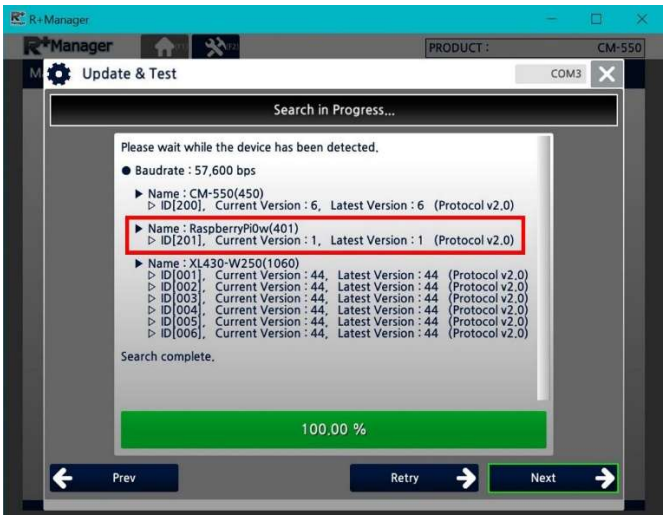


Fig. 3.2 MANAGER shows connection to the RPi Zero W as DXL(201).

3.1 Using MANAGER for Pi Camera

Let us use MANAGER to obtain an overview of the functionalities of the "RPi0W+Camera", a.k.a. Dynamixel 201.

Fig. 3.4 lists the basic functionalities/addresses that a programmer would need to use:

- **Address 11** or RPi **Mode**:
  - **Standby** (= **0**), used to terminate any other RPi Mode currently active.
  - **Color Detection** (= **1**), used to find a given color **set at Address 55** or **Sub-Mode** (7 preset colors from Red to White). The detected color blob also has its area properties computed and available to the programmer via Addresses 60, 62 and 64 (see Fig. 3.5).
  - **Face Detection** (= 2), used to detect a face along with its gaze's direction, to determine which eye is open and whether the face is smiling, also to provide the eyes distance.
  - **Video Streaming** (= 3), used to stream video from the Pi Camera to a Mobile Device running the ENGINEER App (at very slow frame rates).
  - **Marker Detection** (= 4), used to recognize the ROBOTIS 2-D markers (No. 1 to 6). Addresses 60-64 also report on the Marker's area properties and Address 66 provides the Detected Marker Number determined by the RPi Marker Recognition algorithm (see Fig. 3.5).
  - **Line/Lane Detection** (= 5). Currently, only the first two Line Angles in degrees are reported in Addresses 68 and 69, respectively.

3.2 Using TASK

As previously mentioned in Section 2.1.7, the ENGINEER App has a new Hi-Res Display mode for its Touch/Display screen when programming in TASK or MicroPython, i.e. the CM-550 programmer can now access the native pixel resolutions of the Mobile Device's Graphics Display at the SMART **Addresses 10460** (i.e. Screen Width = [0-65535]) and **10462** (i.e. Screen Height = [0-65535]).

The reader may also recall from Section 2.1.7 that when the Mobile Display is used as an Output Device for a possible "Text", "Shape" or "Number" Item, the programmer needs first to "prepare" a **32-**

bit integer **Screen Position Parameter** (SPP) which has the combined information from 4 separate parameters, of **8 bits** each, designated as the said Item's **Position No, Item No, Size No** and **Color No**. When using the Low-Res Display option, Parameter **Position No** can take on values between 1 and 25.

When using the **Hi-Res Display** option, the programmer needs to follow a **3-step** procedure:

1) **Assign 0** to Parameter **Position No**.
2) Assign SMART Parameters **Display Position X** (Address **10480**) and **Display Position Y** (Address **10482**) with appropriate native pixel row and column coordinates.
3) Then the programmer can issue SMART **Display Text/Shape/Number** commands as normal.

3.2.1 Hi-Res Display of Random Finger Touches

In this project "PTC_SD_HelloWorld_RandomTouch_HiRes.tsk3", the goal is to display Text Item 1 "Hello World!" in Hi-Res mode and in **random Colors/Sizes** wherever the user happens to press on the Mobile Display with **one or two** fingers.

The program "PTC_SD_HelloWorld_RandomTouch_HiRes.tsk3" uses the CM-550's UART port (BT-210) to act as the **REMOTE Port** and as the **TASK PRINT** Port back to the Desktop PC. While the embedded BT-410 (**APP Port**) is used to communicate with the ENGINEER App running on the Mobile Device.

3.2.2 Wheel Synchronization & Maneuver Compensation with IMU

The Pan-Tilt Commando (PTC) robot uses a total of six XL430-W250 (single-servo) actuators: servos with IDs from 1 to 4 are configured in Wheel mode and servos with IDs 5 and 6 are configured in Position Control mode. Back in Section 2.1.5, the author demonstrated how to synchronize Goal Position commands to multiple actuators using the Synch-Write procedure. For synchronization of multiple actuators in wheel mode, we can use a different technique based on the concept of "Secondary/Shadow ID" at Address 12 of the XL430-W250 Control Table (https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#secondaryshadow-id12).

Let's recall that an actuator's Primary ID is unique within the Dynamixel Network of a ROBOTIS system and it is represented by Parameter ID at Address 7 (https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#id7). For example, in Fig. 3.1, the back servo on the right of the PTC robot has a Primary ID of 3 at Address 7, but it can also be assigned a Shadow ID of 1 at Address 12. From then on, whenever the TASK program issues a Goal Velocity command to ID=1, both servos 1 and 3 move together in synchronization, but if the TASK program issues a Goal Velocity command to ID=3, only servo 3 would move.

The project "PTC_RC_IMU_SD_RPi_VS.tsk3" illustrates the combination of "**Sync-Write**" and "**Shadow-ID**" techniques to the Wheel Servos (ID = 1 to 4), and for Servos 5 and 6 the use of the Embedded IMU to autonomously keep the Pan-Tilt platform oriented in its original direction, even when the user remotely controls the robot wheeled chassis to go into other directions. Furthermore, the Pi Camera is used in Web Streaming mode to the Mobile Device to serve as a visual check of the run-time performance for the IMU-based maneuver-compensating algorithm used.

We will go through the details of the local Functions used in this project first. Fig. 3.9 shows Part 1 of Function **Init()** of "PTC_RC_IMU_SD_RPi_VS.tsk3" whereas parameters within the EEPROM area of the XL430-W250 actuators are modified:

- Line 146 turns Torque/Power Off on all actuators so that their EEPROM areas can be modified.
- The first FOR LOOP (Lines 147-150) sets the Drive Mode (Address 10) to 0, i.e. Normal mode and Velocity-Profile Control for Servos 1 to 6.

- The second FOR LOOP (Lines 151-154) sets the Operating Mode (Address 11) to 1, i.e. Velocity Control (Wheel) mode for Servos 1 to 4.
- Line 155 synchronizes Servo 3 to Servo 1 by setting its Shadow ID (Address 12) to 1.

- Please note that this procedure works even better with more wheel servos used on each side of the robot!

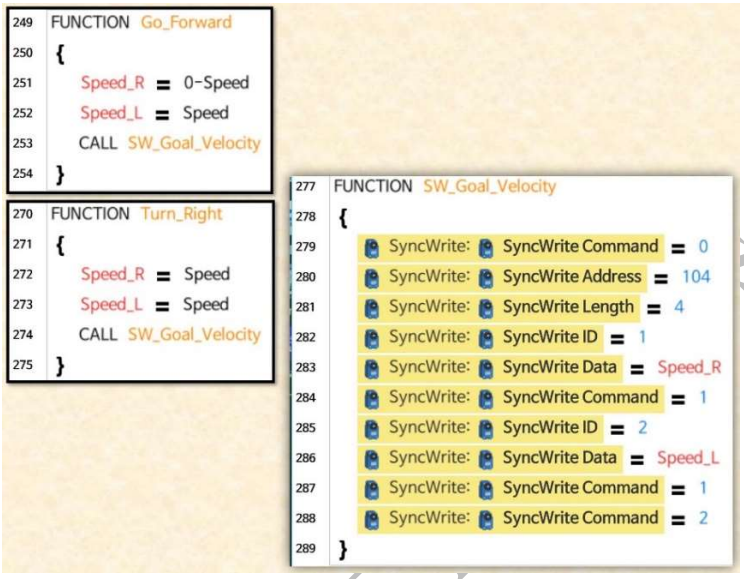


Fig. 3.12 "Go_Forward" and "Turn_Right" maneuvers for "PTC_RC_IMU_SD_RPi_VS.tsk3".

Fig. 3.13 shows that another SyncWrite scheme is used to set Goal Positions for Servo 5 and Servo 6 for two example actions "Right_Pan" and "Up_Tilt" with the Pan-Tilt platform:

- Function **Right_Pan()** shows that it would decrease the current **Pan_Position** (Servo 6) by 5 units each time that it is called during run-time. However, **Pan_Position** has an upper limit of **1396** that Servo 6 cannot go beyond and additionally Function **Alarm_GP()** would be called to sound off a buzzer signal for this situation (see Fig. 3.14).

Similarly, Function **Up_Tilt()** shows that it would increase the current **Tilt_Position** (Servo 5) by 5 units each time that it is called during run-time. However, **Tilt_Position** has an upper limit

---

3.2.3 Pi Camera: Color Detection & Visual Servoing/Ranging

In this project, the Pi Camera is put into its Color Detection mode (ID[201]:Address 11 = 1). When in this mode, the Pi Camera provides live data regarding the XY camera coordinates of the Mass Center of the Tracked Color-Blob and its size (i.e. number of pixels occupied by this blob). Unfortunately, there is no facility for the Pi Camera to display the live image that it is processing at the same time, i.e. no visual confirmation for the programmer/operator.

The goal of the program "PTC_RC_VSR_SD_RPi_CD.tsk3" is to use the Color-Blob's Center Coordinates and its Pixel Size to control the Pan-Tilt platform to keep the Blob centered within the Pi Camera View Port and also to maneuver the PTC robot's wheels in an appropriate direction to keep the Blob's Pixel Size within a range of its Original Size, when the Blob was first captured. This program uses an overall logical structure similar to the one used for "PTC_RC_IMU_SD_RPi_VS.tsk3", except

that the IMU compensation task is replaced with a **"simple"** Visual Servoing/Ranging (VSR) task using the Pi Camera in Color Detection mode. Thus, the common code sections will just be mentioned in passing.

Fig. 3.23 shows Part 2 of the Main Function for "PTC_RC_VSR_SD_RPi_CD.tsk3":

- For some reasons, the author had found that resetting the Pi Camera mode (Line 25), before setting the Pi Camera to its Color Detection mode, created some run time problems for the author's code, thus Line 25 was commented out. Thus, if needed to, the reader can uncomment Line 25 for his/her needs.

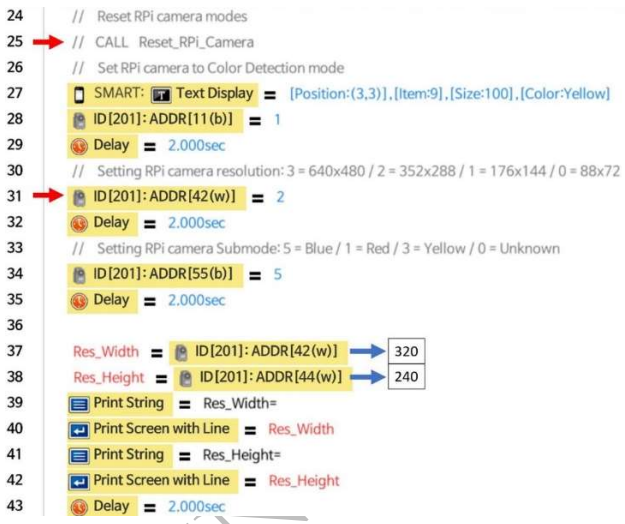


Fig. 3.23 Part 2 of Main Function in "PTC_RC_VSR_SD_RPi_CD.tsk3".

- Line 28 sets the Pi Camera to its Color Detection mode, and the 2 second time delay (Line 29) is needed to let the RPi0W and Pi Camera to "settle down" in this new operating mode.
- Line 31 sets the Pi Camera's resolution to 352x288, but the actual resolution obtained is only 320x240 (as reported by Lines 37-38). The author had also tried the 640x480 mode, but still only the 320x240 mode is obtained at run time.
- Line 34 set the Pi Camera's **Sub-Mode** to "**5**", i.e. for **Blue** Color to track.

3.2.4 Color Track Following Using Pi Camera

Fig. 3.31 shows how the author configured the Pi Camera as a Color Line Detector whereas Servo 5 can be used to "tilt" the camera's viewport closer (best) or further away from the front wheels.

There are two options for using the Pi Camera as a Color Line Detector:

1. Option 1 is to set the Pi Camera into its Color Detection mode (i.e. Address 11 set to 1) and to make use of the X-coordinate of the Mass Center of the Blob that comprises the portion of the Color Track that happens to be within the Pi Camera's viewport at runtime. In this mode, the Pi Camera's maximum image resolution is 320x240 pixels.
2. Option 2 is to use the Pi Camera's own Line Detection mode (i.e. Address 11 set to 5). In this mode, the Pi Camera's maximum image resolution is 160x120 pixels. Internally, this image is further divided into two halves: Zone 1 is closer to the robot's front wheels and Zone

2 is further ahead of the robot (see Fig. 3.31). Each zone is then processed separately to yield two "Line Angles" (Addresses 68 and 69) that represent an overall direction of the Color Track within each zone:

a. If the Color Track goes straight forward ahead of the robot, these Line Angles would have numerical values close to 90 degrees.
b. If the Color Track diverges to the left of the robot, these Line Angles would have numerical values greater than 90 degrees.
c. If the Color Track bends to the right of the robot, these Line Angles would have numerical values smaller than 90 degrees.

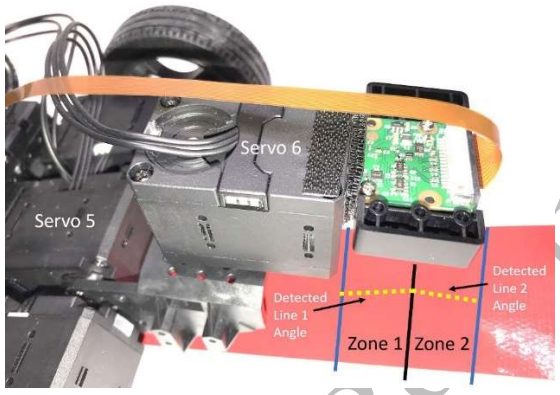


Fig. 3.31 Pi Camera's Setup for Color Line/Track Detection.

3.2.5 Scheduling Maneuvers Using ROBOTIS QR Markers

The ENGINEER Kit 2 also provides 6 QR-type markers which are implemented with the project "PTC_RC_SD_RPi_MKR.tsk3" presented herein:

- Marker 1 stands for "Start".
- Marker 2 stands for "Forward".
- Marker 3 stands for "Backward".
- Marker 4 stands for "Left Turn".
- Marker 5 stands for "Right Turn".
- Marker 6 stands for "Delay 1s".

To put the Pi Camera into the Marker Detection mode, its Address 11 needs to be set to 4 (see Line 27 in Fig. 3.43), then at runtime when a legitimate marker is presented to the Pi Camera at a distance around 30 cm or 12 inches, the programmer can check on 4 Parameters (which should be greater than 0 at that time) via a TASK Custom Command:

- Addresses 60 and 62 for the X-Y Screen Coordinates of the Mass Center for the Detected Marker. A screen resolution of 320x240 pixels is used in the Marker Detection mode.

3.2.6 Dual-Bot RC Using ZigBee Broadcast

This project illustrates how to use ROBOTIS **ZigBee Broadcast** technology (https://emanual.robotis.com/docs/en/parts/communication/zig-110/#nn-communication) to allow the control of two CM-550 robots from a single TASK Output Monitor window on the Desktop PC (see Fig. 3.50):

- The first robot is the PTC with Pi Camera set to Video Streaming to a Mobile Device while being remotely controlled and it is also capable to send back sensor data to the PC on demand.
- The second robot is the MonoBot which is only programmed to be remotely controlled but able to send sensor data back to the PC on demand.

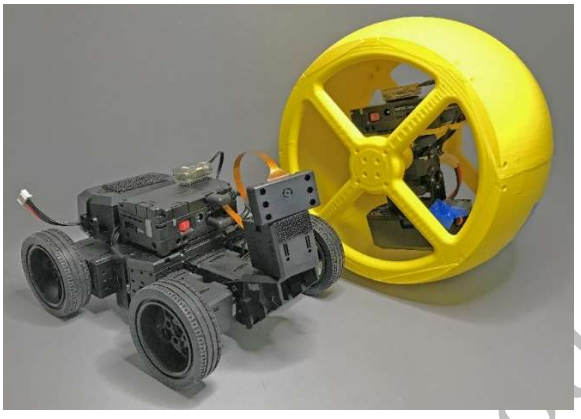


Fig. 3.50 PTC with Pi Camera and MonoBot.

Although the ROBOTIS ZigBee technology dated back from c. 2005, designed for use with the CM-5 and CM-510/530 controllers, it is still the only broadcast wireless technology from ROBOTIS at present. However, some of its components such as USB2DYNAMIXEL and ZIG-110A are discontinued since c. 2019. Consequently, it may be hard for some readers to repeat this project, **but the concepts described here in apply to all ROBOTIS controllers as it is based on the Remocon Packet**. Hopefully in a few years, ROBOTIS would release a new BT module based on Bluetooth V. 5, supporting Mesh Networks, which would boost the runtime performance of this project.

3.3 Using MicroPython

As previously mentioned in Section 2.2.7, **only the ENGINEER App can use a new Hi-Res Display** mode for its Touch/Display screen when programming in MicroPython (and TASK): i.e. the CM-550 programmer can now access the native pixel resolutions of the Mobile Device's Graphics Display at the SMART **Addresses 10460** (i.e. Screen Width = [0-65535]) and **10462** (i.e. Screen Height = [0-65535]).

The reader may also recall from Section 2.2.7 that when the Mobile Display is used as an Output Device for a possible "Text", "Shape" or "Number" Item, the programmer needs first to "prepare" a **32-bit** integer **Screen Position Parameter** (SPP) which has the combined information from 4 separate parameters, having **8 bits** each, designated as the said Item's **Position No, Item No, Size No** and **Color No**. Lastly, when using the Low-Res Display option, Parameter **Position No** can take on values between 1 and 25.

However, when using the Hi-Res Display option, the programmer needs to follow a 3-step procedure:

1) **Assign 0** to Parameter **Position No**.
2) Assign SMART Parameters **Display Position X** (Address **10480**) and **Display Position Y** (Address **10482**) with appropriate native pixel row and column coordinates.
3) Then the programmer can issue SMART **Display Text/Shape/Number** commands in the same way as for the Low-Res case.

3.3.1 Hi-Res Display of Random Finger Touches

In this project "PTC_SD_HelloWorld_RandomTouch_HiRes.py", the goal is to display Text Item 1 "Hello World!" in Hi-Res mode and in **random Colors/Sizes** wherever the user touches on the Mobile Display with **one or two** fingers.

The program "PTC_SD_HelloWorld_RandomTouch_HiRes.py" uses the CM-550's UART port (BT-210) to act as the **REMOTE Port** (CM-550's Address 43) and as the **TASK MONITOR Port** (CM-550's Address 35) to connect to the Desktop PC. While the embedded BT-410 (**APP Port** – CM-550's Address 36) connects to the ENGINEER App running on the Mobile Device (see Lines 8-10 in Fig. 3.64). Please note that DXL(200) refers to the CM-550 Controller as a programming object.

```
5  from pycm import *
6
7  def init_comm_3():
8      DXL(200).write8(43, 1)   # Set Remote Port to UART
9      DXL(200).write8(36, 0)   # Set App Port to BLE
10     DXL(200).write8(35, 1)   # Set Task Monitor Port to UART
11
12 # Main program starts here
13 console(UART)   # BT-210 is used on the UART port
14 init_comm_3()
15 print("PTC is ready!")  # print to CONSOLE
16 # Initialize Text Display Control Parameters
17 nPosition = 0    # Use 0 to access Hi-Res Display
18 nItem = 1   # i.e. "Hello World!"
19 nSize = 0
20 nColor = 0
```

Fig. 3.64 Initialization Steps used in "PTC_SD_HelloWorld_RandomTouch_HiRes.py".

Please note that a similar code segment is needed for the user's Second Touch (i.e. working with Paramters **touch2_X** and **touch2_Y**), which is included in the actual Python code, but it is not shown in this Section 3.3.1. Fig. 3.67 shows a typical run-time result on the Mobile Display.

Fig. 3.67 Run-time Mobile Display obtained for "PTC_SD_HelloWorld_RandomTouch_HiRes.py".

We will go through the details of the local Functions used in this project first. Fig. 3.68 is a listing of Function **Init()** of "PTC_RC_IMU_SD_RPi_VS.py" whereas parameters within the EEPROM and RAM areas of the XL430-W250 actuators are modified:

- Line 10 turns Torque/Power Off on all actuators so that their EEPROM areas can be modified.
- For Servos 1 to 6, the first FOR LOOP (Lines 11-12) sets their Drive Modes (Address 10) to 0, i.e. Normal mode and Velocity-Profile Control. Please note the MicroPython difference in

setting the upper limit for the loop counter "i" (i.e. **7**), as compared to TASK coding (i.e. **6** - see Fig. 3.9).

- For Servos 1 to 4, the second FOR LOOP (Lines 13-14) sets their Operating Modes (Address 11) to 1, i.e. Velocity Control or Wheel mode.
- Line 15 synchronizes Servo 3 to Servo 1 by setting its Shadow ID (Address 12) to 1.
- Line 16 synchronizes Servo 4 to Servo 2 by setting its Shadow ID (Address 12) to 2.
- For Servos 5 and 6, the third FOR LOOP (Lines 17-18) sets their Operating Modes (Address 11) to 3, i.e. Position Control mode for Servos 5 and 6.
- Line 19 sets Address 44, i.e. **Velocity Limit = 250** for the servos in Wheel Mode (i.e. Servos 1 to 4).
- Line 20 reestablishes power to all servos, i.e. to save the new settings into the EEPROM area. From now on, only parameters in the RAM area can be modified safely.
- For Servos 1 to 4, the fourth FOR LOOP (Lines 22-23) sets the Goal Velocity (Address 104) to 0, i.e. stops the PTC robot.

```
 9  def init():
10      dxlbus.torque_off()    # Start modifying EEPROM
11      for i in range(1, 7):  # Set all DXLs to Normal Drive
12          DXL(i).write8(10, 0)
13      for i in range(1, 5):  # Set DXLs 1 to 4 to Wheel mode
14          DXL(i).write8(11, 1)
15      DXL(3).write8(12, 1)  # Setting Shadow ID (=1) for DXL 3
16      DXL(4).write8(12, 2)  # Setting Shadow ID (=2) for DXL 4
17      for i in range(5, 7):  # Set DXLs 5 to 6 to Position Control mode
18          DXL(i).write8(11, 3)
19      DXL(254).write32(44, 250)  # Set Velocity Limit for all DXLs
20      dxlbus.torque_on()    # end modifying EEPROM
21
22      for i in range(1, 5): # Set Goal Velocity for DXLs 1 to 4
23          DXL(i).write32(104, 0)
24      for i in range(5, 7): # Set Acceleration & Velocity Profiles for DXLs 5 and 6
25          DXL(i).write32(108, 0)
26          DXL(i).write32(112, 75)
27          DXL(i).write32(116, 2048)  # Set Goal Position for DXLs 5 and 6
```

Fig. 3.68 Function **Init()** for "PTC_RC_IMU_SD_RPi_VS.py".

Fig. 3.78 defines various image-related parameters that are used in the VSR algorithm:

- The gray rectangle on the right panel in Fig. 3.78 represents an image captured by the Pi Camera and the blue rectangle within it represents the "target area" where the VSR algorithm would strive to keep the "target object" within it at run-time by using the Pan-Tilt platform (i.e. Servos 5 and 6) and when necessary to maneuver the whole robot via its wheels (i.e. Servos 1 to 4).
- Please note that the Pi Camera sets the Bottom Left Corner of its Image Frame as the Origin of the X-Y coordinate axes.
- Line 143 defines Parameter **x1** which marks the centerline of the captured image.
- Lines 142 and 144 define respectively Parameters **x1** and **x3**. Together these 3 parameters divide the captured image into Zones A, B and C as shown in Fig. 3.78.
- At run-time, if the "target object" is located within Zone A or Zone B, the PTC robot would use its wheels to maneuver itself (left or right) so that the "target object" is brought into Zone C. This is the first type of robot control actions.
- For the second type of robot control actions, if the "target object" is found to be in Zone C, then the Pan-Tilt platform is activated (Left/Right and Up/Down) to keep the "target object"

within the blue "target area" using Parameters **object_WL**, **object_WR**, **object_WD** and **object_WU** defined by Lines 145-148.

- As a third type of robot control actions, the PTC robot can also go forward or backward to keep the object's total pixel area within a range defined by Parameters **object_Area_Low** and **object_Area_High** (Lines 152 and 153).

```
133 def init_VSR():
134     global res_Width, res_Height, x1, x2, x3, object_WL, object_WR, object_WD, object_WU
135     global object_Area_Low, object_Area_High
136     # Setting up Parameters used for Visual Serv
137     res_Width = DXL(201).read16(42)
138     res_Height = DXL(201).read16(44)
139     print("Res_Width = ", res_Width)
140     print("Res_Height = ", res_Height)
141     delay(2000)
142     x1 = res_Width / 4
143     x2 = res_Width / 2
144     x3 = res_Width - x1
145     object_WL = x2 - 30
146     object_WR = x2 + 30
147     object_WD = (res_Height / 2) - 30
148     object_WU = (res_Height / 2) + 30
149     object_X_0 = DXL(201).read16(60)
150     object_Y_0 = DXL(201).read16(62)
151     object_Area_0 = DXL(201).read16(64)
152     object_Area_Low = object_Area_0 - 100
153     object_Area_High = object_Area_0 + 100
```

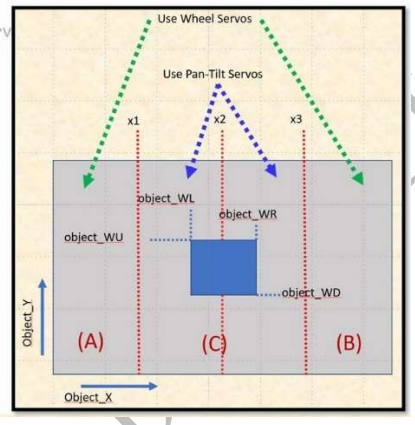


Fig. 3.78 Parameters used in the VSR Algorithm of "PTC_RC_VSR_SD_RPi_CD.py".

Fig. 3.89 lists the code section (within the Main Endless Loop) that would be executed once Button 5 (i.e. **MKR_set**) or Button 6 (i.e. **MKR_reset**) is tapped by the user at run time:

- Lines 233-242 are executed if **(MKR_set > 0)** – i.e. when RC Button 5 is tapped:
  - Text Item 13 ("Marker Node On") is displayed in Yellow on the Mobile Device's Screen at Position (3, 3) (Line 233).
  - **MKR_Control** is set to TRUE (Line 234).
  - Line 235 starts playing Melody 0 out of the CM-550 buzzer (5 seconds duration), but please notice that **buzzer.wait()** is pushed down to Line 241, so that the intervening statements are executed while the music is being played.
  - Line 236 stops the robot.
  - Lines 237 and 238 initialize Parameters **marker_current** and **marker_index_max** to the value of **-1**. They will be later used in Functions **add_marker_list()** and **run_marker_list()**.
  - The FOR LOOP (Lines 239-240) initializes all 8 elements of the ByteArray **markers[]** to zero.
  - Lastly, **MKR_set** is reset to 0 (Line 242): this step is done to make sure that Lines 233-242 are executed only once per user's push on RC Button 5.
- Similarly, Lines 244-247 are also executed ONCE if **(MKR_reset > 0)** – i.e. when RC Button 6 is tapped:
  - Text Item 0 (i.e. "nothing") is used to clear the Mobile Display of any previous Text Item displayed at Position (3, 3) - (Line 244).
  - **MKR_Control** is set to FALSE (Line 245).
  - Line 246 stops the robot.
  - Lastly, **MKR_reset** is reset to 0 (Line 247): this step is done to make sure that Lines 244-247 are executed only once per user's push on RC Button 6.

```
229         if (data_in == 0):
230             stop()
231         else:
232             if (MKR_set > 0):   # Part 1 of Mark
233                 smart_display_text(13,13,100,6)
234                 MKR_Control = True
235                 buzzer.melody(0)
236                 stop()
237             → marker_current = -1   # initial
238             → marker_index_max = -1
239                 for i in range(8):  # i varies
240                     markers[i] = 0
241                 buzzer.wait()
242                 MKR_set = 0
243             elif (MKR_reset > 0):
244                 smart_display_text(13,0,100,2)
245                 MKR_Control = False
246                 stop()
247                 MKR_reset = 0
```

Fig. 3.89 Code Sections responding to User's Push on Buttons 5 or 6 in "PTC_RC_SD_RPi_MKR.py".

Please also note that Fig. 3.95 indicates that the user can push on several RC-100 keys at the same time during run time, as each of these keys are "parsed-out" separately (Lines 176-183).

```
185         if (data_in == 0):
186             stop()
187         else:
188         # Robots RC control for UDLR
189             if ((All_robots > 0) or (Robot1 > 0)):
190                 if (forward > 0):
191                     go_forward()
192                 elif (backward > 0):
193                     go_backward()
194                 elif (left > 0):
195                     turn_left()
196                 elif (right > 0):
197                     turn_right()
198
199             # Sensor monitoring
200             if ((Sensor > 0) and (Robot1 > 0)):
201                 yaw_Robot1 = int((imu.yaw() / 100))
202                 print("Robot1 Yaw = ", yaw_Robot1)
203                 # DXL(200).write16(59, yaw_Robot1)  # 59 = TxD Address
204                 rc.write(yaw_Robot1)
205                 print("!_*_!")
206                 delay(500)
207                 Sensor = 0
```

Robot1_Yaw=22
?U??!_*_!
Robot1_Yaw=4
?U??!_*_!

Fig. 3.96 IF-ELSE and IF-ELSE-IF structures used to determine robot's actions in "PTC_BRC_ZGB_SD_RPi_VS.py".

Fig. 3.96 shows that to make the PTC robot perform a maneuver, the user has to combine a Button among the usual **U-D-L-R Button** with either **Button 1** to make (Robot1 > 0) **OR Button 4** to make (All_Robots > 0) (Line 189 of Fig. 3.96).

The IF-ELSE-IF structure represented by Lines 190-197 indicates that only 1 Maneuver Direction among F/B/L/R is allowed at any one time.

3.4 Using Standard Python

In this Section 3.4, the author's goals are to use Standard Python to enhance the existing capabilities of the CM-550 as illustrated in Section 3.2 and 3.3 by using a Desktop PC as a Supervisory or Direct Controller to the CM-550 and also to implement a USB webcam as an alternate solution to the Pi Camera.

3.4.1 USB Camera: Color Detection & Visual Servoing/Ranging

Using a wired USB camera is not ideal with a mobile robot like the PTC robot as the "wire" may get entangled with the robot itself and therefore prevents its proper operation. But a wired USB camera is readily available to most users, thus the author developed this project with the goal of comparing its runtime performance with the one obtained with the Pi Camera in Section 3.3.3.

Fig. 3.104 shows the PTC with a Logitech C270 attached to the Pan-Tilt platform with the **unused** Pi Camera moved to the Controller area behind the Logitech web camera. The C270 camera is set to the 640x480 pixel resolution mode, i.e. 4 times the resolution of the Pi Camera when it is in Color Detection mode (see Section 3.3.3). The C270 is controlled by a Standard Python program running on a Windows Desktop PC using the OpenCV package (V. 4.4.0.42).

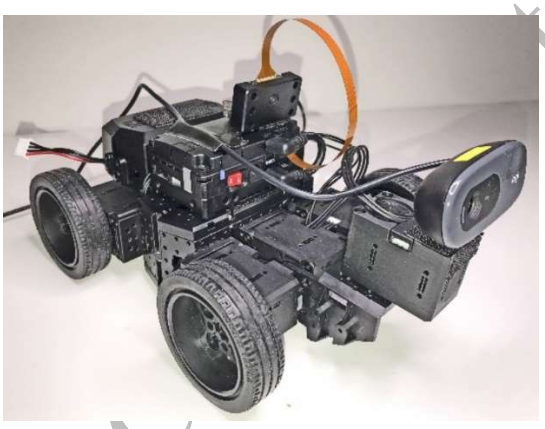


Fig. 3.104 PTC Robot with Logitech C270 Web Cam in "PTC_RC_Color_Tracker_XY.py".

This configuration allows the exploration of two robot control approaches:

1. The first approach is to recreate an operating environment equivalent to the one used with the Pi Camera in Sections 3.2.3 and 3.3.3, thus the PC does the Target/Object Detection work and sends the resulting data such as Target Area and Target X-Y coordinates, via Remocon packets, over to the PTC Robot which would then make its own appropriate Visual Servoing and Ranging tasks. Essentially, the Desktop PC is acting as a glorified Camera Sensor to the CM-550. The resulting programs from this first approach are named "PTC_RC_Color_Tracker_XY.py" and "PTC_RC_VSR.py/tsk3".

Please note that, **for at least every THREE iterations of the Main Endless Loop, the VSR Algorithm is activated ONCE** (controlled by Lines 264-266 and Line 232), during which the Pan/Tilt Platform AND/OR the PTC Wheel Platform may be activated so as to keep the Color Target within the Image Frame's Center.

```
244             #  Adjusting pan-tilt platform
245             if ((object_X > x1) and (object_X <= object_WL)):
246                 left_pan()
247             elif ((object_X >= object_WR) and (object_X < x3)):
248                 right_pan()
249             elif ((object_Y < res_Height) and (object_Y >= object_WD)):
250                 down_tilt()
251             elif ((object_Y <= object_WU) and (object_Y > 0)):
252                 up_tilt()

253         # Adjusting robot wheels
254         if ((object_X > 0) and (object_X <= x1)):
255             turn_left()
256         elif ((object_X >= x3) and (object_X <= res_Width)):
257             turn_right()
258         elif ((object_Area >= object_Area_Low) and (object_Area <= object_Area_High)):
259             stop()
260         elif ((object_Area > object_Area_High)):
261             go_backward()
262         elif ((object_Area < object_Area_Low)):
263             go_forward()
264         object_X = 0
265         object_Y = 0
266         object_Area = 0
```

Fig. 3.119 Part 5 of Main Endless Loop in "PTC_RC_VSR.py".

3.4.2 Dual-Bot RC Using ZigBee Broadcast

In this project, the ROBOTIS ZigBee Hardware (in Broadcast mode) is applied once again between the Desktop PC and the two Robots PTC and MonoBot (please review Section 3.3.6 if needed). However, the TASK Output Monitor Tool used in Section 3.3.2 will be replaced by a Standard Python application written to act as a more general-purpose ZigBee Central Station capable of sending and receiving Remocon Packets to and from both robots.

The resulting Python solutions are named "PTC_MB_ZGB_Central.py" for the PC, "MB_BRC_ZGB_ID.py/tsk3" for the MonoBot, and "PTC_BRC_ZGB_SD_RPi_VS_ID.py/tsk3" for the PTC Robot.

The requirements/features of the program "PTC_MB_ZGB_Central.py" are listed below:

1. The Operator uses the PC keyboard as the Robot Control Interface:
    a. **Key 1** is programmed to be PUSH-ON/PUSH-OFF to indicate whether Robot1 (PTC) is under the Operator's control.
    b. **Key 2** is also programmed to be PUSH-ON/PUSH-OFF to indicate whether Robot2 (MonoBot) is under the Operator's control.
    c. **U/D/L/R Arrow** keys are used in conjunction with **Key 1** or **Key 2** to move the **specified robot(s)** (either one or both) in the usual maneuver directions F/B/L/R. To accommodate the special control requirements of the MonoBot (see Fig. 3.101), **up to 2** Arrow keys can be pressed by the Operator at any one time to control the robots.
    d. **Key 3** is also of the type PUSH-ON/PUSH-OFF and is also used in conjunction with **Key 1** or **Key 2** to trigger a **Sensor Read** mode for either or both robots which would then send their own specified **sensor reads continuously** to the PC (when Key 3 is ON).

Fig. 3.128 describes Part 3 of the Keyboard Processing Procedures used in "PTC_MB_ZGB_Central.py" which is activated when the Operator pushes on Button/Key 1 to select/deselect the PTC robot for subsequent Maneuver and Sensor Read commands:

```
207     if ((key_1 == No_1) or (key_2 == No_1)):  # Toggling on Key 1 (i.e. PTC)
208         if (robot_1 == False):
209             img_1 = cv2.imread(im_PTC_ON)
210             cv2.imshow(window_1, img_1)
211             robot_1 = True
212             m_1 = 1
213         else:
214             img_1 = cv2.imread(im_PTC_OFF)
215             cv2.imshow(window_1, img_1)
216             robot_1 = False
217             m_1 = 0
218         cv2.waitKey(1)   # to refresh display
```

Fig. 3.128 Part 3 of Keyboard Processing Procedures in "PTC_MB_ZGB_Central.py".

Once one or two robots are chosen by the Operator with Key 1 or Key 2, the Operator can next push on Key 3 to activate the Sensor Read mode which is "automatic/autonomous" *(U/D/L/R Keys are still actionable)*. Coding for the Sensor Read mode is listed in Figs. 3.134, 3.135 and 3.136. Fig. 3.134 shows Part 1 of the Sensor Read procedure which is only activated if **(sensor_mode == True)** (see Line 276):

```
275 # Sending & Receiving Sensor-Read Remocon Packet for both robots
276     if (sensor_mode == True):
277         # Preparing TxData to send to robot(s)
278         TxData = 0  # clear out TxData just to be safe
279         TxData |= (m_3 << 6)  # m_3 should be set to 1
280         if ((robot_1 == True) and (robot_2 == False)):
281             TxData |= (m_1 << 4)  # VRC Button 1 pushed
282             send_data()
283             print("Sensor Read command sent to PTC = ", TxData)
284         elif ((robot_1 == False) and (robot_2 == True)):
285             TxData |= (m_2 << 5)  # VRC Button 2 pushed
286             send_data()
287             print("Sensor Read command sent to MonoBot = ", TxData)
288         elif ((robot_1 == True) and (robot_2 == True)):
289             TxData |= (m_1 << 4)  # VRC Button 1 pushed
290             TxData |= (m_2 << 5)  # VRC Button 2 pushed
291             send_data()
292             print("Sensor Read command sent to PTC & MonoBot = ", TxData)
293         time.sleep(0.1)
```

Fig. 3.134 Part 1 of the Sensor Read Procedure in "PTC_MB_ZGB_Central.py".

3.4.3 Dual-Bot Relay-RC Using ZigBee Broadcast

At present, only the ROBOTIS ZigBee hardware allows another interesting communications configuration that offers a "relay" type of network (as the author tries to stay away from using the term "mesh" network offered by Bluetooth V.5, perhaps available in future ROBOTIS communications hardware). Once again, the PC acts as the ZigBee Central Station in broadcast mode. However, one of the two robots used in Section 3.4.2 will act as a Mid/Relay Station (PTC) and will be put near the limit range of the ZIG-100/110 antennas (~10 feet indoors, at least for the author's environment) from the PC. While the second robot (MonoBot) will be located beyond the first ZigBee range, so that the PC can effectively only communicate with the Mid/Relay Station (PTC), but the PTC can communicate with both the PC and the MonoBot. The PTC then needs to have a scheme to pass along packets destined for the PC or for the MonoBot, whenever the PTC happens to receive such packets during its operations. Previously, in Section 3.4.2, PTC was Video Streaming to the ENGINEER Mobile App during its

operations, but this task is removed from PTC's repertoire in this Section, while new features are added as listed below.

The resulting Python solutions are named "PTC_MB_ZGB_Central_Relay.py" for the PC, "MB_BRC_ZGB_Relay.py/tsk3" for the MonoBot, and "PTC_BRC_ZGB_Relay.py/tsk3" for the PTC Robot.

3.4.4 Dual-Bot RC Using Dual BT-210s with PySerial

Fortunately for Standard Python users, the PySerial Module is written to handle the use of multiple RS-232 COM ports simultaneously, so the communications programming approach used in this project is straight forward:

1. For each robot (PTC and MonoBot), use a different BT-210 module connected to the UART Port for each CM-550. The PC's Windows 10 OS already can handle multiple BT-210s easily.
2. Create and Associate COM Device **ser1()** with the PTC robot, and COM Device **ser2()** with the MonoBot.

This project "PTC_MB_DBT_Central.py" is essentially an adaptation of the previous project "PTC_MB_ZGB_Central.py" which was already described in detail in Section 3.4.2. Thus, only the relevant changes are reported in this Section 3.4.4.

Fig. 3.164 shows the needed changes for the usage of the PySerial module:

- Two separate PySerial devices are now needed: **ser1** for the PTC robot and **ser2** for the MonoBot (Lines 83-84).

```
83 ser1 = serial.Serial('COM15', 57600)
84 ser2 = serial.Serial('COM19', 57600)

53 def send_data(robot_no):
54     global TxData, TxD_packet
55     lowbyte = TxData & 0xff
56     TxD_packet[2] = lowbyte & 0xff
57     TxD_packet[3] = ~lowbyte & 0xff
58     highbyte = (TxData >> 8) & 0xff
59     TxD_packet[4] = highbyte & 0xff
60     TxD_packet[5] = ~highbyte & 0xff
61     if (robot_no == 1):
62         if (ser1.write (TxD_packet) != 6):
63             print("Robot1 Transmit Error!!!")
64     elif (robot_no == 2):
65         if (ser2.write (TxD_packet) != 6):
66             print("Robot2 Transmit Error!!!")
67     time.sleep(0.100)
```

Fig. 3.164 Usage of PySerial Module in "PTC_MB_DBT_Central.py".

3.5 Using C/C++

This Section 3.5 is written as a continuation of Section 2.4.3 whereas the ROBOTIS ZigBee SDK was introduced to the reader with the project "SBwA_Color_Tracker_XY.cpp", which only needed to send Remocon packets from the Desktop PC to the CM-550 based robot running either on TASK or MicroPython codes. In Section 3.5, we will explore more concepts and techniques in shaping the 16-bit Remocon Message to contain more information regarding actuators and sensors in the transmission and reception of single or multiple Remocon Packets, between the Desktop PC and one or two CM-550 robots, and using ZigBee and Bluetooth communications hardware.

Section 3.5 is also written as a parallel development of the concepts and techniques developed in Section 3.4 for Python, but now they will be translated/adapted into C/C++ codes: for example, Section 3.4 used PySerial as an alternative to using the ROBOTIS ZigBee SDK, while Section 3.5 will be showcasing the use of the Boost.Asio library. Thus, if the reader happens to get to this Section 3.5 from Section 2.4, the author is apologizing that he will need to refer the reader back to some materials developed back in Section 3.4.

Fig. 3.168 illustrates the robots used in developing projects in this Section 3.5: "Pan-Tilt Commando" (PTC) with a Logitech Webcam C270 (as an alternate solution to the Pi Camera) and "MonoBot".

- Please note the important Line 355 which sets **stop_once** to TRUE which makes the above procedure execute **only once for each time** that the Operator releases all Keys.

```
350         // When all keys released send "0" packet ONCE
351         if ((key == -1) && (stop_once == false))
352         {
353             cout << "Sending ZERO packet ONCE" << endl;
354             send_data(0);  // send "0" packet
355             stop_once = true;
356         }
357     }   // end of IF track_mode == 0
358     // Tracking for chosen object & sending RC_Packet to CM-550
359     else if (track_mode == 1)
360     {
361         // OPENCV SECTION
362         vid_cam >> src; //read current video frame
363         // Convert src (BGR) to hsv
364         cvtColor(src, hsv, COLOR_BGR2HSV);
365         split(hsv, HSVchannels); // Splits the image into 3 channel images
```

Fig. 3.173 Last Procedure of Standard RC mode used in "PTC_RC_Color_Tracker_XY.cpp".

```
373         if (target_moments.m00 >= 30)
374         {
375             target_x = target_moments.m10 / target_moments.m00;
376             target_y = target_moments.m01 / target_moments.m00;
377             target_area = target_moments.m00 / 5000;
378             cout << "Target Center X = " << target_x << endl;
379             cout << "Target Center Y = " << target_y << endl;
380             cout << "Scaled Target Area = " << target_area << endl;
381             // Preparing TxData (comm_Type + target_x or target_y or target_area)
382             // Sending target_x data
383             TxData = 0;
384             comm_Type = 1;
385 Packet 1    TxData = (comm_Type << 14) | (int(target_x) & 0x3fff);
386             send_data(TxData);
387             cout << "Target X sent\n";
388             // Sending target_y data
389             TxData = 0;
390             comm_Type = 2;
391 Packet 2    TxData = (comm_Type << 14) | (int(target_y) & 0x3fff);
392             send_data(TxData);
393             cout << "Target Y sent\n";
394             // Sending target_area data
395             TxData = 0;
396             comm_Type = 3;
397 Packet 3    TxData = (comm_Type << 14) | (int(target_area) & 0x3fff);
398             send_data(TxData);
399             cout << "Scaled Target Area sent\n";
400         }
401         else  // target data not valid enough, so stop the robot
402         {
403             send_data(0);
404         } // end of if-else (target_moments.m00 >= 30)
```

Fig. 3.174 Procedures used to send Target Pixel Area and Screen Coordinates X/Y to PTC in "PTC_RC_Color_Tracker_XY.cpp".

3.5.2 Dual-Bot RC Using ZigBee Broadcast

If the reader is not familiar with the ROBOTIS ZigBee technology, he/she is referred to Section 3.3.6 which should be read first, before going on to this Section 3.5.2 and the next Section 3.5.3.

In this project, the ROBOTIS ZigBee Hardware described in Section 3.3.6 is applied between the Desktop PC and the two Robots PTC and MonoBot. This C/C++ project uses a single COM port to communicate with both CM-550 robots using ZigBee Broadcast mode.

This project's goal is to compare its run-time performance to its "mirror" project written in Standard Python (Section 3.4.2). The resulting C/C++ solution is named "PTC_MB_ZGB_Central.cpp" for the PC side and it is designed to work with the previously developed CM-550 codes: "MB_BRC_ZGB_ID.py/tsk3" for the MonoBot, and "PTC_BRC_ZGB_SD_RPi_VS_ID.py/tsk3" for the PTC Robot.

The overall code structure developed for "PTC_RC_Color_Tracker_DC.cpp" in Section 3.5.1 can be ported over to this project "PTC_MB_ZGB_Central.cpp", but additional codes will be needed to handle Remocon packets that would be **sent back from the robots to the PC**.

Fig. 3.184 shows the initial preparation to prepare and send a typical Maneuver-type Remocon packet to PTC, i.e. **robot_1 == TRUE** on Line 304:

- TxData is first cleared to zero for robot_1 (Line 306).
- Message bit **m_1** is shifted left 4 bits and inserted into **TxData** (Line 307).
- Function **prepare_data()** is next called (Line 308). This function adds on the other message bits as needed into **TxData**. The details for this function are provided in Figs. 185 and 186.
- Upon return from Function **prepare_data()**, if **TxD_OK** is set to TRUE (Line 309), signifying that the final **TxData** message is ready to be sent to **robot_1**, Function **send_data()** is called to assemble the complete 6-byte Remocon Packet and to broadcast it out to all robots. The details for Function **send_data()** are provided in the previous Fig. 3.171.
- A similar procedure is coded for MonoBot (**robot_2**) but using message bit **m_2**.

```
303     // Preparing Maneuver - type Remocon Packet for Robot1 = PTC
304     if (robot_1 == true)
305     {
306         TxData = 0;  // clear out TxData just to be safe
307         TxData |= (m_1 << 4); // Start with VRC Button 1 added wi
308         prepare_data();
309         if (TxD_OK == true)
310         {
311             send_data(TxData);
312             cout << "TxData sent to PTC = " << TxData << endl;
313         }
314     }
```

Fig. 3.184 Inserting Robot 1 ID into Remocon Message in "PTC_MB_ZGB_Central.cpp".

- If Line 419 turns out to be FALSE, i.e. Tilt or Velocity mode is already selected, then the ELSE branch at Line 438 is taken, and **pan_mode** is simply reset to FALSE by Line 439.

```
417     if ((key_1 == p) || (key_2 == p) || (key_1 == P) || (key_2 == P))
418     {
419         if ((tilt_mode == false) && (velocity_mode == false))
420         {
421             if (robot_1 == true)
422             {
423                 if (pan_mode == false)
424                 {
425                     putText(img_1, "Pan Set", Point(65, 115), FONT_HER
426                     imshow(window_1, img_1);
427                     pan_mode = true;
428                 }
429                 else
430                 {
431                     img_1 = imread(im_PTC_ON);
432                     imshow(window_1, img_1);
433                     pan_mode = false;
434                 }
435                 waitKey(1);   // to refresh display
436             }
437         }
438         else
439             pan_mode = false;  //  Key p/P pushed by mistake
```

Fig. 3.189 Image and Text procedures for PTC's Pan mode in "PTC_MB_ZGB_Central_Relay.cpp".

3.5.4 Dual-Bot RC Using Dual BT-210s with BOOST.ASIO

In this last C/C++ project, the BOOST.ASIO library is used. The author used Version 1.74.0 (https://www.boost.org/doc/libs/1_74_0/doc/html/boost_asio.html), and specifically its "Serial Ports" devices (https://www.boost.org/doc/libs/1_74_0/doc/html/boost_asio/overview/serial_ports.html).

For readers new to the BOOST libraries, the author recommends Mukherjee (2015) for a good reference to help getting started with various aspects of software installation and usage on Windows and Linux machines (also see Appendix A for the author's tips).

The basic procedure for using a BOOST Serial Port in C/C++ is as follows:

1. Request an I/O service from the OS with a statement such as **io_service io_1;**

   for more information visit this web link (https://www.boost.org/doc/libs/1_74_0/doc/html/boost_asio/overview/core/basics.html).

2. Instantiate a new "serial_port" device connected to a COM port specified by the user and associate it to the previous I/O service line named **io_1**, for example:
   a. **const char* PORT1 = "COM24";**
   b. **serial_port ser1(io_1, PORT1);**
   c. from then on, use BOOST.ASIO functions to set up the communications options such as Baud Rate, and then use Write and Read functions as needed:

      serial_port_base::baud_rate BAUD(57600);

      ser1.set_option(BAUD);

      write(ser1, buffer(TxD_packet, 6)); // "buffer" is an ASIO function

      read(ser1, buffer(RxD_packet_1, 6);

for more information please visit
https://www.boost.org/doc/libs/1_74_0/doc/html/boost_asio/overview/serial_ports.html

---

3.6 Wrapping up Volume 1

The author considers Volume 1 to be a "foundational" volume and hopes that the robotics concepts and programming techniques showcased will be found "useful" by the readers. The organization of this Volume is unusual for reasons already mentioned in Chapter 1 (Section 1.3) and the author also hopes that the readers did not find it confusing or too much repetitive.

The ROBOTIS ENGINEER Robotic System with larger memory and more communications ports on the CM-550, along with more control features for its Dynamixel Actuators and RPi based camera, provides a big leap in performance and functionality in the Educational Robotics area. In Volume 1, the author has explored only some of its main functionalities and would like to "share" his "lessons learned":

1. The addition of the MicroPython functionality to the CM-550 earns a Big "LIKE" from the author as it allows standard structures like arrays and other data types than "integer", and also other standard language usage for function calls and object programming. It also allows for more compact coding.
2. The MicroPython Interpreting Engine has a surprisingly good runtime performance as compared to TASK codes which are essentially compiled C codes. But the ROBOTIS MicroPython Editor was clunky to use considering the author's editing habits, so he preferred to use the Thonny's IDE for editing and the ROBOTIS MicroPython Editor just for "compiling" and "download".
3. The CM-550 expanded COM Ports programming features, along with the use of the "undervalued" Remocon Packet protocol, allow "greater mileage" to the ENGINEER system (as well as the older ROBOTIS systems – *meaning that users can mix them all up*), as they allow versatility in adding co-controlling computing/sensing platforms such as Desktops and Laptops (Volume 1), as well as Single-Board-Computers based on Windows or Linux OSes (planned for Volume 2). The author had sent his recommendation to ROBOTIS folks to expand their Remocon Packet protocol to be able to handle 32 bits for the actual data, i.e. a 10-byte packet instead of the current 6-byte packet (in existence since 2005!) – we'll see if ROBOTIS would take on such recommendation.
4. Not too surprisingly, C/C++ provided the best runtime performance but Python provided quicker development times.