Rune: Robot-User Nexus

Senior Design Project

Final Technical Report

Nicolas C. Ward

May 5, 2005

Abstract

Rune, the Robot-User Nexus, is the culmination of two years of software development work for the Swarthmore Robotics Team. The two goals of this ongoing project are to create a robust and extensible software architecture for mobile robots, and to create an intuitive, flexible, and extensible interface for mobile robot teleoperation. This report describes a beta version of Rune that is documented and ready for limited distribution and testing.

Contents

List of Tables

List of Figures

1 Introduction

This project is the culmination of nearly two years of work as a member of the Swarthmore Robotics Team. During that time, I developed two successive versions of a teleoperation user interface. I also contributed small updates and general code improvements to a number of other software modules that are a part of the Swarthmore Robotics Team codebase.

The further development of this work does not consist solely of a contribution to the Swarthmore Robotics Team; it is our hope that much of this code will be used by researchers and developers at other institutions. As such, a significant portion of this Senior Design Project consists of preparing our system for distribution. This preparation includes both further code development and significant documentation efforts.

Fundamentally, this project is about moving from alpha to beta, that is from an unstable, unreleasable alpha version of the software to a mostly stable, releasable beta version. This idea forms the basis of the project goals.

1.1 Goals

1.1.1 Team Goals

The single overarching goal of the Swarthmore Robotics Team is to *create a robust and extensible software architecture for mobile robots*. This idea has influenced nearly all of the design decisions made by team members over the last few years. Such a system would not merely be for internal use; we want to be able to distribute this system to anyone who is trying to do mobile robotics research but who does not have the time to develop their own software architecture.

All of our software will be distributed using an open source license. While we do believe that our system is approaching the point where it is worth sharing with the world, we do not suffer from any delusions that our software is a perfect solution. It is our hope that, as other people begin using our system, they will be willing to make their own contributions, such as finding or fixing bugs, or suggesting or adding new features. None of this would be possible if our software were distributed as a black box.

By the end of the project, we hope to freeze a stable version of our code that is ready for widespread distribution and testing.

Finally, if our software is to be understood by those outside groups using it, then our system must be well-documented before it is distributed. All of the software modules and their interactions must be thoroughly explained in a set of interlinked manuals, developer guides, and user guides.

1.1.2 Project Goals

My primary goal for this project is to *create an intuitive, flexible, and extensible interface for mobile robot teleoperation.* Successfully reaching this goal requires the completion of many tasks. Some of these requirements have already been met as a part of the work that I did over the past

two summers, which is discussed further in Section 2.1 on Page 7.

This interface must also be tightly integrated into the overall system architecture. This is not to say that the architecture should be dependent on the interface, but rather that the interface should be compatible with any and all software modules that conform to the architecture standards.

In addition to creating an interface that could be used by the Swarthmore Robotics Team (and others) as a part of a larger project, I also want to develop a system that would be useful for performing human-robot interaction (HRI) research. There are currently a lot of unanswered questions about the relative importance of the different types of information presented to an operator in a mobile robot teleoperation task. If the interface is sufficiently flexible, that is, if the design and layout of the interface is highly configurable, then my project could be used to perform HRI experiments where specific interface elements could be easily varied for each test subject.

As a subset of one of the Team Goals, I intend to document as many aspects of the interface as possible. This includes inline code comments which may help guide future developers, the code reference that is included in the appendices of this report, and the configuration and extension manuals that I am also including in the appendices.

1.2 Tasks

The task of teleoperating a mobile robot is currently one of the most difficult problems being approached by experts in HRI. This task can only be solved by a system that has all three of these critical components:

1. A simple and intuitive interface
2. A robust communication system
3. A modular software architecture

The first component is the only portion of the system that should be presented to the human teleoperator. The second component significantly improves the performance of the task by the robot-user pair by providing real-time information exchange between the operator and their robot. The third component is needed by the software developers, so that they can easily extend the system to add new functionality or adapt the system to operate on a wide variety of robot and interface hardware.

As a member of the Swarthmore Robotics Team, my focus for the past two years has been on the development of a highly configurable mobile robot teleoperation graphical user interface (GUI) that satisfies the first criterion. This project continues that work.

The other current student member of the team is Fritz Heckel, a Computer Science major in the Class of 2005. He has developed the higher-level portions of our communication system, to ensure that our architecture satisfies the second criterion. He has also developed a library of standard messages and functions that we now use to ensure that all of our software literally speaks the same language when our modules try to communicate. This interoperability is an important aspect of the third criterion.

As all of our modules are currently under development, it is very important that we maintain the modules such that they remain interoperable. When this project is completed, we expect to freeze all development at a releasable version so that our codebase has a stable foundation for future development.

1.3 Definitions

Before continuing with this report, there are some important terms that need to be defined. This section should help clarify some of the terms used to explain information in the body of the report.

1.3.1 Graphical User Interface

A Graphical User Interface, commonly abbreviated as GUI, is one of the primary methods by which a human can interact with a computer. A GUI at its most minimal consists of one or more display devices and one or more input devices. Although the use of modern personal computers is now widespread, any discussion of interface design requires a fairly low-level understanding of user interface components.

The display device typically consists of some sort of hardware screen, such as a cathode ray-tube (CRT) or liquid crystal display (LCD) monitor, a digital light projection (DLP) or LCD projector, a light-emitting diode (LED) or LCD text display, or some array of status LEDs. The screen display device is the method by which the computer conveys information to the user.

The input device typically consists of a coupled pointing device and tactile array. The pointing device moves a cursor in two-dimensions to navigate the various on-screen displays. A pointing device, such as a mouse, trackball, joystick, or drawing pad, usually provides context to other user input by selecting some subset of the GUI. A tactile array consists of a set of buttons or switches, such as a keyboard. Each button is associated with some input symbol which the computer can interpret.

A GUI is a simple and powerful way for converting user intentions into actions that can be strictly interpreted by the computer as specific input events, and for converting the abstract data representations that a computer may use internally into something that is meaningful to a human user.

1.3.2 Human Robot Interaction

Human-robot interaction (HRI) takes many forms, depending on the specific application. For this project, I am only considering the set of human-robot interactions that involve an operator remotely controlling or giving commands to a robot by using some sort of GUI.

HRI is a specialized subset of the more general field of human-computer interaction (HCI). Because this project centers on the development of a powerful interface for robot teleoperation, awareness of standard HRI and HCI techniques are extremely useful in making design decisions.

HRI studies of multiple versions of this interface, along with studies that cover other groups' teleoperation systems, have given me a lot of information about how to approach the task of developing

Table 1: Mobile Robot Platforms

Robot	Count	Manufacturers	Locomotion
Magellan Pro	3	Real World Interface (RWI)	wheeled
Blimp	1	Geoff Hollinger '05 Alex Flurie '05 Zach Pezzementi '05	flying
ROV	1	Maila Sepri '05 Samantha Brody '05	submarine

Rune. Those contributions are covered in more detail in Related Work (Sec. 2, pp. 7).

1.3.3 Mobile Robot

A mobile robot is literally a robot that can move. The method of locomotion is variable. For our purposes, I will refer only to the types of robots used by the Swarthmore Robotics Team. Those robots, how many of them the team has, who made them, and their type of locomotion are specified in Table 1 on Page 4.

1.3.4 Situational Awareness

Situational awareness (SA) is a fairly general term that refers simply to an individual's ability to perceive what is going on around them. In the case of HRI, particularly in the context of robot teleoperation, I will use the following definition from [1]:

> The understanding that the humans have of the locations, identities, activities, status, and surroundings of the robots.

Measuring the quality of an operator's SA requires some sort of metric. For this project I use a fairly standard one that uses six independent factors: task effectiveness, neglect tolerance, robot attention demand, free time, fan out, and interaction effort [2]. These terms are defined in Table 2 on Page 5.

It is worth noting that Neglect Tolerance is basically a measure of the autonomy of a robot. Also, Free Time and Robot Attention Demand are complements of one another. That is, $FT = 1.0 - RAD$.

These terms, although more relevant to an HRI study than to the software development that is part of this project, are important to understanding some of the ways in which the quality of such a system is evaluated.

Table 2: Definitions of HRI SA Metrics

Metric	Definition
Task Effectiveness	How well was the task completed by the operator?
Neglect Tolerance	How much time passed after the operator stopped interacting with the robot before the robot passed its minimum effectiveness threshold?
Robot Attention Demand	How much of the operator's attention does the robot require?
Free Time	How much time does the operator have for tasks other than operating the robot?
Fan Out	How many robots can the operator effectively use at once?
Interaction Effort	How much of the operator's cognitive and physical resources are dedicated to the task of operating the robot?

1.3.5 Swarthmore Robotics Team

The Swarthmore Robotics Team is an interdisciplinary group led by Dr. Bruce A. Maxwell, a professor in the Swarthmore Department of Engineering. Over the last several years, the student members of the team have come primarily from the Engineering and Computer Science departments.

During its existence, the team has pursued several major projects, including:

- Indoor Aerial Robot
- Robot Host
- Robot Soccer
- Robot Urban Search & Rescue (USR)
- Underwater ROV

The team has participated in a number of competitions relevant to these projects, performing admirably in all of them and achieving victory in several of them. Most recently, Swarthmore was awarded First Place in the mobile robot Urban Search & Rescue competition at the American Association for Artificial Intelligence (AAAI) 2004 Conference in San Jose, CA. The team set an all-time record high score for the competition, defeating entries from (among others) MITRE and Xerox PARC.

None of these achievements would have been possible without the tens of thousands of lines of code that make up the Swarthmore Robotics Team codebase. All team members have contributed

to this code, which consists of several important software modules and libraries. This architecture is described in greater detail in Section 3 on Page 13.

1.3.6 USR Test Arena

The USR Test Arena is maintained by the National Institute of Standards and Technology (NIST) as a standard reference for comparing the performance of robotic USR solutions. The test arena is used at all official USR competitions, and is available at the NIST site and at other locations worldwide.

The test arena consists of three separate arenas, each of which is scaled to a different difficulty level. These arenas were all designed and constructed by NIST's Manufacturing Engineering Laboratory.

- The Yellow Arena is navigable by almost all robots. The floor is flat, there are few obstacles, and only minimal debris.
- The Orange Arena is slightly more difficult. There are sections containing raised flooring, other sections that are covered with small debris, and an elevated section with a ramp and a stairwell representing a collapsed upper floor.
- The Red Arena is navigable only by legged, wheeled, whegged, or snake robots. Large debris covers the entire arena, which represents the rubble pile of a collapsed building.

Because our wheeled RWI Magellan Pro robots are intended only as research robots, they can only handle the Yellow Arena, and small clear floor areas of the Orange Arena. A significant investment in hardware would be required for the Swarthmore Robotics Team to be able to participate in the event more fully.

Recently, NIST added a new venue: the Black Arena, jokingly referred to as the "Reality Arena". The Black Arena is an unmodified abandoned Nike missile site with victims distributed arbitrarily within. The site is prone to flooding, and has numerous stairwells, access tunnels, ductwork, ramps, and the like. This arena is the gold standard of USR tests [3]. This arena will not be accessible by the robot members of the Swarthmore Robotics Team for some time.

1.4 Report Organization

I have separated this report into two parts: the first being a research paper describing the work I have done as part of the Swarthmore Robotics Team, focusing on the portion that is my Senior Design Project, and the second being a set of appendices that document the configuration, use, and extension of Rune, the Robot-User Nexus.

The Related Work section (Sec. 2, pp. 7) positions the work I've done as part of the Swarthmore Robotics Team in the context of the wider fields of human-robot interaction, human-computer interaction, and graphical user interface design. It focuses on the inspirations for the project, how we integrated those ideas into the design of the overall system, and some of the major design decisions that I made. It also covers the history of this project up until the spring of 2005.

The System Architecture section (Sec. 3, pp. 13) describes the current set of software and hardware solutions used by the Swarthmore Robotics Team, as well as the communication between those components.

The Rune section (Sec. 4, pp. 16) provides information about the role and overall functionality of Rune, the teleoperation interface that I have developed.

The Appendices (pp. 23) contain the detailed output from Doxygen, an automatic code documentation utility created by Dmitri van Heesch [4]. They also contain full documentation on how to configure Rune, and how to add functionality to Rune.

2 Related Work

2.1 Project History

2.1.1 Before 2003

The first system used by the Swarthmore Robotics Team, developed in 2001 and 2002, consisted of a simple GUI display that provided camera video and map data for up to two robots, along with a set of clickable buttons that would send camera or movement commands to the appropriate software modules running on those robots. Unfortunately, this system was essentially unusable in a competition environment due to lags in communication.

A backup to this system was also used in the competition. Video broadcasts were displayed in a single window, using the `dispipc` utility to receive data from the vision module. Commands were issued to the robot using multiple remote text terminals running command-line shells. In each shell, the command-line test utility for each module was used to send commands. The problems with this system were well documented by a group researching HRI at the competition [5].

This system, while functional, was only usable by a person thoroughly experienced in the development of the system. Even for an experienced individual, the system was not terribly responsive, and required typing out commands with explicit numerical values. It was difficult to use, slow to respond, and did not meet the basic requirements of situational awareness.

2.1.2 Summer 2003

The version of the interface that I designed in 2003 was a totally new set of code. Some of the design elements were inspired by previous versions. I dubbed this version Robobot, for no reason other than that our version control archive needed a name. Unfortunately, this tongue-in-cheek christening lasted for over a year.

The fundamental design was related to the previous version from 2002, but there were some major differences. First, the GIMP ToolKit, GTK+, was used as the widget library for displaying the GUI components. The previous version had used Motif, a much older and less flexible widget library. In terms of usage, GTK+ allowed me to develop an interface that could take command input from

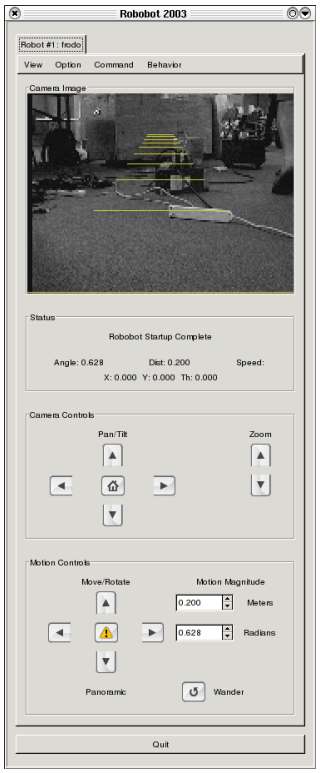


Figure 1: Pre-Alpha: Robobot 2003 – Fully Expanded Interface View

fixed keyboard bindings, or from interface button events. A menubar for toggling options, such as image size, as well as multiple text displays to report status, were included.

The full version of the interface is shown in Figure 1 on Page 8. The primary aspect of the interface is still the image from the robot's onboard camera. The options menubar is placed above the image. Below the image, in several sections, are buttons which can be clicked to issue translate and rotate commands, and to reorient the pan/tilt/zoom (PTZ) camera.

A more streamlined version of the interface is shown in Figure 2 on Page 9. None of the buttons are displayed. In this mode, the operator is expected to have memorized the keymaps that are used to activate the same functionality as the buttons. We chose to use the standard WASD configuration for movement, as it is common in many 3-D video games. For the Magellan, W was bound to forward movement, S to reverse, A to counter-clockwise rotation, and D to clockwise rotation.

I was the operator for all of our USR competition runs at the 2003 International Joint Conference on Artificial Intelligence (IJCAI). I used this collapsed interface view exclusively, since I had memorized all of the necessary key bindings. I found that the more limited view actually had more functionality, because I could react more quickly using the fix key bindings, and because the screen real estate was less cluttered. Other users could use a menu item to toggle between the two views.

In addition to the primary interface, a separate application, XRoboview, was used to monitor the robot's range sensors and the orientation of the robot's PTZ camera. XRoboview, whose only window is shown in Figure 3 on Page 9, was originally designed as a testing and debug program

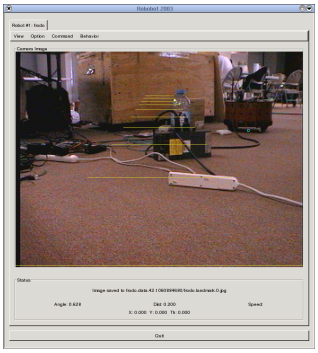


Figure 2: Pre-Alpha: Robobot 2003 – Collapsed View

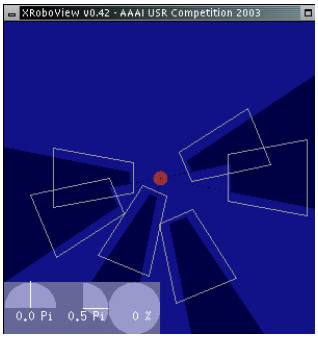


Figure 3: Pre-Alpha: XRoboview

for the new Nav module that Fritz Heckel was beginning to develop. When I tried to integrate the range sensor display into Robobot, I found that GTK+ could not update the graphical display with a high enough frequency to be useful. This required that a Robobot and XRoboview window both be open, in addition to the terminal windows that spawned them.

In retrospect, the GTK+ library is really intended to develop GUIs for non-real time applications, such as editors, utilities, web browsers, and the like. Although its widget library is extensive, most of them are more useful for the manipulation of text or data and not of images. It was an improvement over Motif, in that the library is easier to use and results in cleaner code, but because of its function as a wrapper for the X windowing system, GTK+ is much slower than a raw X window.

The results of the IJCAI ’03 competition were promising. The Swarthmore Robotics Team placed 2nd, behind the team from the Idaho National Labs (INEEL). INEEL’s robot, an RWI ATRV Jr., was a superior hardware platform, with greater ground clearance, more powerful motors, and a very expensive sensor package that included a laser rangefinder and a FLIR (forward-looking infrared) camera. Their system allowed them to enter more difficult areas of the test arena, finding victims in the Orange and Red Arenas that earned more points.

Although we did not win, the competition was definitely a success. We had a significant lead in points over the next team, and were capable of quickly finding victims in the Yellow Arena.

2.1.3 Winter 2004

In February of 2004, Fritz and I had the opportunity to take our robots down to the NIST (National Institute of Standards and Technology) site in Gaithersburg, MD to participate in an HRI usability study of USR teleoperation interfaces. The study was run by Jean Scholtz from NIST, Jill Drury from MITRE, and Holly Yanco from the University of Massachusetts - Lowell [5]. The team from INEEL was also invited to test their interface.

The NIST USR Test Arena, when it is not traveling to a robotics conference for a competition, is set up in an abandoned bunker on the NIST site. In addition to the engineered difficulties of the course, our Magellan robots faced a new environmental condition: cold temperatures. We were plagued by numerous hardware failures, including a brake relay broken by the cold and battery operating cycles of less than 15 minutes.

The subjects of this study were all first responders (EMTs, firefighters, etc.), but none of them had had any robot teleoperation experience. Although our system barely worked due to the temperature problems, I learned a lot about what parts of the current interface design were lacking.

2.1.4 Summer 2004

The idea for the new version of Robobot was first conceived on the back of a napkin at an Italian restaurant in Vienna, Austria. At the time, it consisted only of the relationships between widgets, viewports, and visualizers, and how the fit together inside of their parent view. This overall object hierarchy was implemented in Robobot, and still exists in Rune today, although the actual implementation is considerably more complicated. This part of the architecture is discussed thoroughly in Section 4 on Page 16.

I implemented this version of Robobot using the Simple DirectMedia Layer library, an open source graphics library intended for use in video games. It is fast and responsive, and contains support for USB joysticks. An example screenshot, from during the competition at the 2004 American Association for Artificial Intelligence conference, is shown in Figure 4 on Page 4.

Fritz developed GCM and Robomon over this summer as well, which significantly changed the communication layer of our architecture, as discussed in Section 3.1 on Page 13. Robobot was an integral part of these changes, since they both grew together. Fritz also developed a new Nav module, and Prof. Maxwell added image compression capabilities to SVM that markedly increased our video framerate. Since we have found that the video is so central to the use of Robobot and Rune, this was a very important improvement.

It would not be out of line to claim that the Swarthmore Robotics Team destroyed the competition in the AAAI ’04 USR event. The new Nav module made our old little Magellans fast; Robomon and its management abilities made our system more stable and fast to recover from a crash; the SDL library, combined with the new compression in SVM, made the entire interface much more responsive. We were getting closer and closer to our goals.

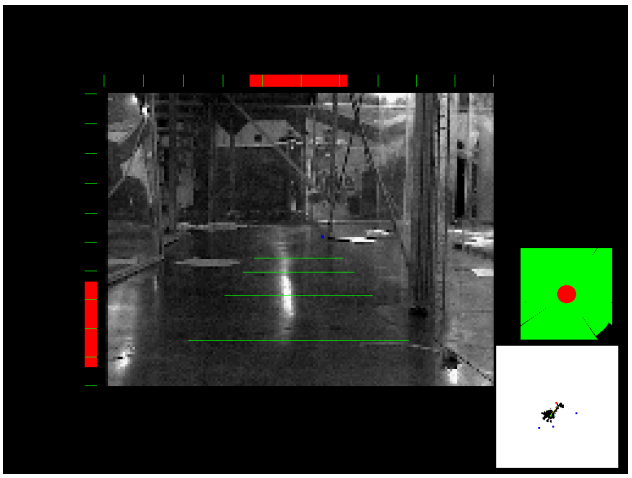


Figure 4: Alpha: Robobot 2004

2.1.5 Fall 2004

Most recently, as part of his CS thesis, Fritz made several changes to Robomon. The most important of these was the addition of capabilities, which define what a robot can do in a more general and more granular fashion than modules. These new developments are covered in Section 3.3 on Page 15.

2.2 Inspiration

The design of Rune is a synthesis of ideas from several different sources. I spent most of 2003 really learning what the whole task was about. While I did successfully develop a teleoperation interface, it was certainly not the best configuration. We had the opportunity to be exposed to several other interface designs, as well as to meet several people who do significant research in the field of HRI.

I started development from scratch in 2004, with new ideas about how Robobot should look, how it should function, how it should be configured, and how it should be controlled. All of those influences are discussed in this section.

2.2.1 Look and Feel

The primary design inspiration for the user interface comes from first-person shooter (FPS) video games, such as *Unreal Tournament* [6]. We have explored this paradigm thoroughly in our recent publications [7][8].

While it may seem unusual to take cues from a game that consists largely of killing virtual people and apply ideas to a case where we try to save people, the tasks are fundamentally the same. In the game, as in a simulated USR environment, the user is searching a complex three-dimensional

environment for targets while attempting to achieve a goal.

In addition to the similarity between their respective tasks, FPS games also demonstrate the state of the art when it comes to real-time interfaces for interacting with a 3-D virtual environment. Players want their in-game characters to be as carefully controlled as their own physical movement in reality.

Because of our desire to make the system function as much like a video game as possible, the video from the robot's onboard camera has been the primary interface element throughout all of the versions of the interface, starting with the simplest designs from four years ago. Humans rely heavily on visual input to navigate our world, so we believe that it makes sense for us to rely heavily on visual input to teleoperate a camera-equipped robot in a remote environment.

2.2.2 Control

As discussed in Section 1.3.1 on Page 3, a GUI requires some sort of tactile interface. In my observations of interface use by non-developers during the winter of 2004 (Sec. 2.1.3, pp. 10), most of the study participants strongly preferred the joystick-based interface used by the team from INEEL to our keyboard- and mouse-based interface.

For many people, the joystick is a much more intuitive interface. Moving the joystick in a direction produces the expected movement from the robot. At one of our robot demonstrations held during the Fall of 2004, a small child under the age of six learned how to operate the robot, and within 5 minutes was explaining to adults how to use the joystick element of the interface. This was the point at which we knew that we had chosen the correct direction.

2.2.3 Philosophy

The mantra of this project has been abstraction. Rune's code uses object-oriented programming techniques to maintain a consistent set of operations on a structure by structure basis. The point is to, as much as possible, leave the user and even the developer at the high level of widgets, viewports, and visualizers. The lower level aspects of the code should only need to be understood by those individuals who wish to add new functionality.

Rune is also meant to be extremely configurable. Rune's entire state is defined by an external XML configuration file. All of the relationships between widgets, handlers, visualizers, robots, capabilities, and all of the other objects are defined in that file. This means that, with even the limited number of defined widgets and visualizers we have already, there are a tremendous number of possible interface layouts and configurations.

The configurability idea was inspired in a large part by the work of Holly Yanco, particularly that which has focused on developing better USR interfaces [9]. Her research and the work of Jean Scholtz and Jill Drury has given me, and the Swarthmore Robotics Team in general, a lot of useful input regarding the design of HRI GUIs and in particular those used in USR.

3 System Architecture

As described in Section 1.2 on Page 2, there are three components that are critical to a functional teleoperation system. In this section, I will describe the communication layer and the methods by which it connects and manages all of the other modules in the system, including the interface module. Rune itself is described in greater detail in the next section, Section 4 on Page 16. The physical robots will also be discussed briefly. The overall architecture is based on the REAPER system, also developed at Swarthmore [10].

3.1 Communication Layer

3.1.1 CMU IPC

The communication between all of the software modules is conducted using CMU IPC, an inter-process communication library developed at Carnegie Mellon University [11]. IPC uses sockets, so it can connect software modules on a single computer or across a network. IPC passes messages that contain packed data structures that are broadcast to all modules connected to the IPC server and subscribed to the message type. However, the message formats must be defined by the modules before they can be used.

One of the biggest problems we have had in using IPC has been in cross-platform configurations. When processes that run on different architectures try to send fixed length messages, data can come out of order due to endianness issues, or the data can be packed differently due to the size of the addressable units of memory. We have had intermittent problems in getting Rune running on Mac OS X (on a PowerPC architecture) communicating completely successfully with an instance of `central` running under Linux (on an IA32 architecture).

3.1.2 GCM

In order to standardize the communication between the various modules, and to provide the capability of sending common commands to any module, Fritz developed the General Communication Module (GCM). GCM is not technically a module in its own right, but rather a library that all other modules depend on at compile-time. In addition to defining a common set of messages, GCM also defines the set of available module capabilities for use with Robomon. The GCM library provides a set of utility functions as well, including logging functions, two-dimensional data compression/decompression functions, and basic message handling functions [12].

3.1.3 Robomon

The top-most level of the communication layer is the module control and management module, Robomon. Robomon runs as a daemon on the robot, and is responsible for starting and stopping modules as needed. When Robomon first starts up, it determines what capabilities are available by querying configured modules for their capability listings. As capabilities are requested, the

Figure 5: Rune Communication Layer

Robomon daemon starts and stops the relevant modules. Robomon can also automatically restart modules in the event of a crash, and even replace an unavailable module with a new module that has an equivalent capability set [12].

The entire communication layer, consisting of these three software modules/libraries, can be considered to be a single monolithic module/library, as shown in Figure 5 on Page 14. For convenience and consistency, I will refer to this as Robomon, although technically Robomon is only the topmost part of the layer. From the point of view of Rune, Robomon does appear to be a single module control and management system that provides standardized communication with and control of all software modules in the Swarthmore Robotics Team codebase.

As you can see, this communication layer fulfills the “robust communication” criterion specified in Section 1.2 on Page 2.

3.2 Standard Modules

As the architecture of the overall system in use by the Swarthmore Robotics Team has grown more powerful and robust, so have the individual software modules that do most of the work when the robots are in operation. The modules described in this subsection include a mapping module (Sec. 3.2.1, pp. 15), a navigation module (Sec. 3.2.2, pp. 15), and a vision module (Sec. 3.2.3, pp. 15). When these modules are connected together through the Robomon communication layer, the only thing that they lack is something to control them and give their capabilities a purpose.

Fritz is currently developing a simple control module that has been code-named Pinky. Pinky is intended to function as a basic brain, an autonomous real-time control module that can take input from the other modules, make decisions based upon that input, and direct the actions of the other modules in a cohesive manner. However, since that module is not yet mature, a human operator is still required to perform most actions.

As you will see, all of these modules, when functioning as a unit on a single robot, fulfill the

"modular software architecture" criterion specified in Section 1.2 on Page 2.

3.2.1 Mapping Module

The Swarthmore Mapping Module is the most out of date of all of the software modules running on the robot. Since our physical robots rely on sonar and infrared range sensors to detect obstacles, and use dead reckoning odometry derived from wheel encoders, the map data is unfortunately effectively useless. The module has been made compatible with recent changes in Robomon, and it does have support for capabilities. A new version of the SMM daemon, `smmd`, will be developed over the course of the next year by other students.

3.2.2 Navigation Module

The Swarthmore Navigation Module was completely rewritten last year by Fritz. The new version of the SNM daemon, `snmd`, utilizes the velocity space method [13] of obstacle avoidance. It is a much more responsive navigation module, and it is better suited to use with some sort of autonomy module.

Since Nav was created for use with the RWI Magellan robots, the basic data on which it operates has not significantly changed. It still used Mage to communicate with the robot's rFlex controller board, although a Mage replacement, ArchMage, is currently under development.

3.2.3 Vision Module

The Swarthmore Vision Module has been primarily developed by Prof. Maxwell, although a non-trivial amount of functionality has been added by students over the last few years. I have recently updated SVM to be compliant with the recent changes to Robomon. This included moving some image and two-dimensional data functionality into GCM, which is appropriate given that GCM is where all image-related messages are defined.

I have also made changes to Rune so that it no longer depends directly on SVM but still has the functionality for controlling SVM's behavior through the communication layer. Reducing dependencies is a key aspect of meeting the "modular software architecture" criterion.

3.3 Capabilities

Capabilities are a description of what a given robot (and therefore, a given instance of Robomon) can do. A robot's list of capabilities informs the user what actions that robot can perform, and what data that robot can provide.

Representing the software on a robot in terms of capabilities instead of modules has several useful features. The first is that the representation is more granular: a capability is a very specific thing, associated with only one type of action or data. Contrast this with a module, which is typically a

large piece of software dedicated to an entire category of actions and their associated data. Consider, for example, all of the different types of motion that could be programmed into the navigation module of just a wheeled robot. Add into the mix other robot platforms, and the number of actions that could be performed by each navigation module grows.

The second major feature of capabilities is that they are considerably more general. The functionality of a module tends to change from version to version, as new features are added, unused features are removed, and bugs are fixed. A capability, because it is more granular, can apply to any number of modules that can perform the relevant action or provide the relevant data.

All of the capabilities that are defined in the Swarthmore Robotics Team's architecture are a part of GCM. This ensures that all GCM-compliant software modules can use capabilities, and that a single standard set of capabilities is maintained for internal consistency.

3.4 Other Robots

The application of this system architecture to two totally different hardware platforms, the blimp and submarine, have developed two totally different navigation modules. Unfortunately, there was not time to complete the integration of Rune with these two systems before the academic portions of all three of these Senior Design Projects were due. It is my hope to finish the integration step over the next few weeks so that the Swarthmore Robotics Team can have a complete Land-Sea-Air robot rescue team.

4 Rune

Rune is the Robot-User Nexus, a highly configurable robot teleoperation interface that allows a single human operator to remotely control one or more robots intuitively and efficiently. An alpha version of this interface was tested by me extensively in the robot USR competition at AAAI '04 in San Jose, CA.

If it has not yet been made clear, Rune is the name of the beta version of the Swarthmore Robotics Team's teleoperation interface. Robobot was renamed to Rune mostly to remove all references to a fairly stupidly-named piece of software. The name Rune fits in thematically with a lot of the other software created as part of the team's codebase, such as Mage (which is derived from Magellan, the type of robot it runs on) and ArchMage (the planned successor to Mage, which is both greater than Mage and a "bridge" between systems). Rune can also be considered to stand for its full title, the Robot-User Nexus, literally, the place where the robot and the user come together.

4.1 Objects

As mentioned briefly before in Section 2.2.3 on Page 12, Rune consists of four primary abstract objects. These are the view, the viewports, their visualizers, and their widgets. All of these are discussed in the following subsections.

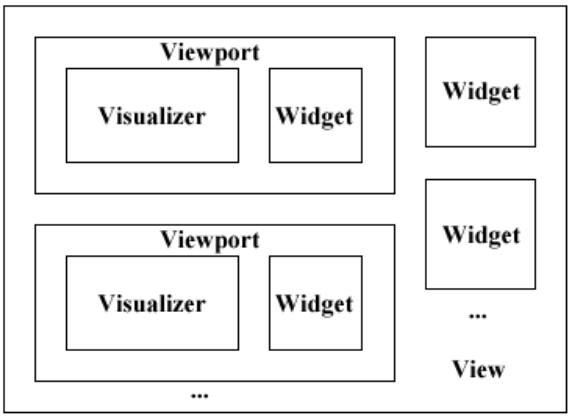


Figure 6: Rune Abstract Object Hierarchy

Technically, Rune's own state object should be the root of the tree. The Rune object keeps track of the main view, some important SDL pointers used in drawing to the screen, such as a text font, and a list of connected robots and joysticks. These aspects of Rune's internal configuration are explained in greater detail in the appendices.

4.1.1 View

The view is only step removed from the root of Rune's object tree, which contains Rune's state. The view is the main window and drawing area, which can either be windowed or zoomed into full screen mode. The preferred mode of operation is full screen, because my experience as a USR operator has shown that an absence of background data improves my perception of the foreground data.

The view can contain any number of viewports, as well as any number of global widgets. There are some logical limitations to this, in the sense that Rune would be fairly useless if it were configured not to display anything. On the opposite extreme, clogging the main view with viewports would both make them all illegible and make the entire interface completely unresponsive as it wasted resources redrawing so many GUI elements.

The global widgets are always active; that is, since they are not associated with a particular viewport, they cannot be disabled except through global state changes. Since most useful widgets need to be able to send messages, as described below, there are very few global widgets. One example would be a quit widget, which is a fairly useful one to have defined. There are no hard-coded widgets of any kind, a reminder of the fact that Rune is completely configurable.

4.1.2 Viewport

A viewport is a single drawable area contained within the view. The area could be as large as the view itself, or cover just a small portion of the view. No viewport is allowed to be completely outside of the bounds of the view, but they can be partially out-of-bounds.

Each viewport is associated with only one of Rune's robot connections. Since each viewport is independent, theoretically the single view could contain data from multiple robots. This feature has yet to be tested, although the architecture is completely defined.

Each viewport can contain some number of widgets. Since a viewport is associated with a robot, the functionality of all of these widgets is generally also associated with that robot. In some cases, the widgets could be used solely to modify the state of the viewport.

A viewport is a fairly abstract data structure. It has a position and size, but most of the work is performed by its child visualizer and widgets. In addition to a viewport's (x, y) position in the parent view, it can also have a z-position. This is essentially a layer number where larger z-values are "more in front" than smaller (or even negative) z-values. This allows viewports to overlap, but to have the order in which they are drawn be well-defined. If the data drawn into a viewport contains an color-keyed alpha channel, then the viewport is composited correctly onto any layers that may be behind it.

4.1.3 Visualizer

A visualizer does the dirty work of converting a specific type of arbitrary data into a representation that can be drawn onscreen, specifically, in the area of the parent viewport. Each visualizer is associated with a type of message by using the message's string name. These messages can be either an IPC message received from the parent viewport's associated robot, or an internal message received from a state-modifying widget.

The generic handler assigned to each message finds matching visualizers, and then passes the message off to those visualizers' associated visualization functions. Each visualization function can draw something, typically a representation of some part of the data contained in the message, into the visualizer's parent viewport.

The list of visualization functions is defined in the source code, as are the visualization functions themselves. As such, adding new types of visualizers requires access to the source code, plus a successful recompilation after the addition.

4.1.4 Widget

Widgets come in two varieties, state-modifying and message-passing, and can be located either within a specific viewport or globally within the view. A widget can both modify state and pass a message, and there is no limit on the number of modifications a single widget handler can make, or on the number of messages that a single widget handler can send.

A widget has a very simple function: when an event that matches its configuration is received, its

associated handler is called on that event. The handler does whatever processing is necessary and then modifies Rune's state or passes a message as appropriate.

A set of utility objects, namely joysticks, controls, and events, are loosely associated with widgets to help define a given widget's set of matching input events. These too are all defined in the configuration file.

State-modifying widgets manipulate some sort of data that is internal to Rune. One example would be a widget that modifies the data structure that is associated with a query-response message. The widget modifies the query so that the response changes appropriately.

Most widgets, however, are of the message-passing variety, since they can send commands to modify the state of the robot according to operator input.

The list of widget handlers is defined in the source code, as are the widget handlers themselves. As such, adding new types of widgets requires access to the source code, plus a successful recompilation after the addition.

4.2 Processes

There are four major processes that operate in Rune. The first is the SDL event loop, which handles user input. The second is one or more IPC listen loops, which handle received messages. The third is the screen refresh loop, which checks for updated visualizers and draws them to their parent viewports, and in turn to the root view and the actual screen buffer. The fourth and final set of processes are those individual timers associated with different capabilities. They are called regularly, as defined by their timeouts, and perform the action that is defined in the source for them.

When an SDL event is popped off of the event stack, its type is checked. If it is a quit event, then Rune cleans up after itself and quits. The quit event can be pushed onto the event stack by the window manager (i.e. with the close window decoration) or by a defined and instantiated quit widget. All other event types are then compared against the events desired by all global and viewport widgets. When a matching widget is found, that event information, along with the matching widget are passed off to that widget's handler, which takes the appropriate action

When Rune receives a broadcast IPC message, the IPC library first checks to see if the interface module is subscribed to that message type. If it is, the message is passed to the IPC handler that subscribed to it, or the handler that was associated with a responding message's query. There are some specialized handlers, such as those that are subscribed to GCM common commands, or Robomon capability listings and module info. The vast majority of messages, however, namely those intended for a visualizer, are passed to a generic fixed length or variable length message handler.

The variable and fixed length message handlers function identically, with the exception of how they handle message data unpacking and message freeing. These functions are naturally different because of the different message formats. The IPC handler determines the robot that originated the message, and then checks all of the visualizers whose parent viewport is associated with that robot. If any visualizer desires the incoming message, it is passed off to that visualizer's visualization function, along with the matching visualizer.

The timer for screen updates is actually associated with Rune's one and only capability, CON_REAL, for real-time control. The main view is only updated if the data in any of the visualizers has changed. Because of the limitations of network bandwidth, this general means the screen update function is running at a much higher frequency that the effective refresh rate. This guarantees that newly visualized information is drawn to the screen as soon as it arrives.

If a visualizer has been updated, the entire screen is wiped, and all of the viewports are re-blitted, that is, copied byte-for-byte, into the screen buffer. The entire screen has to be redrawn to guarantee that all transparency effects are drawn properly.

Finally, we have the assorted event timers that are associated with individual capabilities. These are used to query the modules associated with those capabilities for certain types of data. A query-response model is used here so that network bandwidth is not wasted on the transmission of data that Rune would have to ignore because it was not ready to receive it. Smaller state messages are generally broadcast on a regular basis, but any message whose data structure is larger than a few entries is generally a candidate for query-response instead of broadcast.

5 Future Work

I have made significant progress in the development of the software modules for which I am responsible. I have also contributed to the overall integration of the entire Swarthmore Robotics Team system architecture. There are still interesting avenues of research and design that remain open to future student work.

5.1 Interface Testing

Given the highly configurable nature of Rune, we hope that it will play an important role in future HRI studies. Such studies were an important inspiration (Sec. 2.2, pp. 11) for the design of Rune in the first place, as Rune was designed expressly with HRI researchers in mind.

The first of these experiments will begin in only a few weeks, under the direction of Dr. Holly Yanco and her team at UMass-Lowell. They intend to test a large number of possible interface configurations on subjects of all experience levels. This will allow the HRI community to obtain a lot more data regarding efficient interface designs for robot teleoperation.

5.2 Multi-Robot Teleoperation

Although I successfully operated two robots simultaneously during the USR competition at the AAAI '04 conference, the multi-robot control was achieved by setting up a simple time-sharing hack in the interface. With true multi-robot support now a part of Rune, and with the addition of new robot platforms (the blimp and the submarine), we should soon be able to demonstrate a unified Land-Sea-Air robot team controlled by a single human teleoperator. That achievement would be a truly impressive one, and would clearly demonstrate that the system we have developed is highly flexible and extensible.

The blimp and submarine projects, like most other Senior Design Projects, have just now reached the end of their development cycle. I believe that there would be sufficient motivation to finish the last few integration steps to successfully deploy the Swarthmore Robot Rescue Team.

5.3 Distribution

Once the last few changes are made, Rune will be ready for distribution. With all of the modules cleaned up, we can create a set of install scripts and package the modules and their documentation for distribution from the Swarthmore Robotics Team's website, http://robotics.swarthmore.edu/.

I look forward to seeing how many people choose to try out our software once it is made readily and freely available.

5.4 Updates

It is my intention to maintain Rune for as long as I can, while simultaneously instructing future developers in its use. I understand that Rune, as a piece of software, is not a static creation. I hope that many others will help it grow into something even more powerful and flexible than it already is. I hope that the design philosophy I used when I created Rune remains to leave an impression on those future students who pick up where I have left off.

6 Conclusion

This project, over its two-year duration, has become the dominant aspect of my Swarthmore experience. Although I am happy and proud to see it reach this solid completion point, I know too that I will miss this work.

The first set of long-term goals of the Swarthmore Robotics Team have finally been met. We now have a stable foundation on which future generations of team members can build new and interesting projects. As per the three criteria of this task, we have a robust communication system, a modular architecture, and an intuitive interface.

Rune, and its previous Robobot incarnations, has proven itself repeatedly in use. The Robomon/GCM architecture will allow Rune to continue to be adapted to more and more teleoperated robot systems.

The Robot-User Nexus is ready to accept input.

7 Bibliography

References

[1] J. Scholtz, J. Young, J. L. Drury, and H. A. Yanco, "Evaluation of human-robot interaction awareness in search and rescue," in *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, vol. 3, New Orleans, LA, Apr. 26–01, 2004, pp. 2327–2332. 4

[2] D. R. Olsen and M. A. Goodrich, "Metrics for evaluating human-robot interactions," in *Proceedings of the 2003 Performance Metrics for Intelligent Systems Workshop*. Gaithersburg, MD: National Institute of Standards and Technology, Sept. 16–18, 2003. 4

[3] A. Jacoff, B. A. Weiss, and E. Messina. (2005) Nist reality arena. [Online]. Available: http://www.isd.mel.nist.gov/projects/USAR/Reality%5FArena/ 6

[4] D. van Heesch. Doxygen. [Online]. Available: http://www.stack.nl/%7Edimitri/doxygen/ 7

[5] H. A. Yanco, J. L. Drury, and J. Scholtz, "Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition," *Human-Computer Interaction*, vol. 19, no. 1 and 2, pp. 117–149, 2004. 7, 10

[6] *Unreal Tournament*. Epic Games Inc, 1999. 11

[7] B. A. Maxwell, N. Ward, and F. Heckel, "A configurable interface and software architecture for robot rescue," in *AAAI '04: Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, San Jose, CA, July 25–29, 2004. 11

[8] ——, "A human-robot interface for urban search and rescue," in *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, Aug. 09–15, 2003. 11

[9] M. Baker, R. Casey, B. Keyes, and H. A. Yanco, "Improved interfaces for human-robot interaction in urban search and rescue," in *Proceedings of the 2004 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 3, The Hague, Netherlands, Oct. 10–13, 2004, pp. 2960–2965. 12

[10] B. A. Maxwell, L. A. Meeden, N. S. Addo, P. Dickson, N. Fairfield, N. Johnson, E. G. Jones, S. Kim, P. Malla, M. Murphy, B. Rutter, and E. Silk, "Reaper: A reflexive architecture for perceptive agents," *AI Magazine*, pp. 53–66, 2001. 13

[11] R. Simmons and D. James, *Inter-Process Communication: A Reference Manual*, Mar. 2001. 13

[12] F. Heckel and B. A. Maxwell, "A framework for increased reliability and inter-operability of robot software modules," in *Proceedings of the International Conference on Intelligent Robots and Systems*, Edmonton, Canada, Aug. 02–06, 2005, submitted. 13, 14

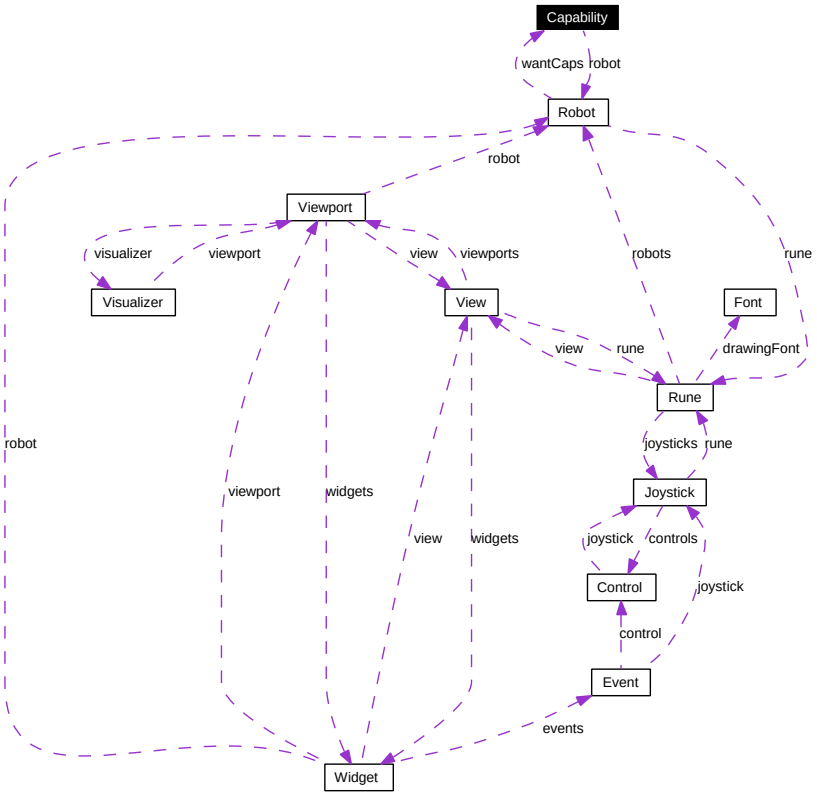
[13] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics and Automation Magazine*, vol. 4, no. 1, pp. 23–33, Mar. 1997. 15

A Rune Data Structure Reference

A.1 Capability Struct Reference

`#include <rune.h>`

Collaboration diagram for Capability:



Data Fields

- **Robot** ∗ **robot**
- GCM_Capability **cap**
- SDL_TimerID ∗ **timers**
- int **nTimers**
- bool **ready**
- void ∗ **state**
- void ∗ **query**

A.1.1 Detailed Description

Defines the configuration for a given robot Capability data structure.

Definition at line 243 of file rune.h.

A.1.2 Field Documentation

A.1.2.1 GCM_Capability Capability::cap

The GCM-defined capability with which this Capability data structure is associated.

Definition at line 252 of file rune.h.

Referenced by parseRobot().

A.1.2.2 int Capability::nTimers

The number of timers in use by this capability.

Definition at line 262 of file rune.h.

Referenced by initCapability().

A.1.2.3 void* Capability::query

Arbitrary data query associated with the capability.

Definition at line 274 of file rune.h.

Referenced by getCapabilityQuery(), initCapability(), parseRobot(), timerRequestImage(), timerRequestMap(), and timerRequestRobotState().

A.1.2.4 bool Capability::ready

Whether or not the capability is available and active on the robot.

Definition at line 266 of file rune.h.

Referenced by parseRobot().

A.1.2.5 Robot* Capability::robot

A pointer to the Capability's parent **Robot**(p. 37).

Definition at line 247 of file rune.h.

Referenced by handleMessageFixed(), handleMessageVariable(), parseRobot(), timerRequestImage(), timerRequestMap(), timerRequestRobotState(), timerSendKeepAlive(), and timerUpdateView().

A.1.2.6 void* Capability::state

Arbitrary state data associated with the capability.

Definition at line 270 of file rune.h.

Referenced by getCapabilityState(), initCapability(), and parseRobot().

A.1.2.7 SDL_TimerID* Capability::timers

An array of IDs of timers associated with this capability. They are optionally used to set a timeout between regular events that **Rune**(p. 40) processes for this capability.

Definition at line 258 of file rune.h.

Referenced by initCapability().

The documentation for this struct was generated from the following file:

- **rune.h**

A.2 CommonRequest Struct Reference

`#include <rune.h>`

Data Fields

- bool **handled**
- GCM_Common_RequestData * **request**

A.2.1 Detailed Description

Defines common request data associated with a capability.

Definition at line 280 of file rune.h.

A.2.2 Field Documentation

A.2.2.1 bool CommonRequest::handled

Whether or not this particular robot state request has been handled.

Definition at line 284 of file rune.h.

Referenced by initCapability(), timerRequestMap(), timerRequestRobotState(), visualizeMapData(), and visualizeRangeData().

A.2.2.2 GCM_Common_RequestData* CommonRequest::request

The robot state request message that will be sent over IPC.

Definition at line 288 of file rune.h.

Referenced by initCapability(), timerRequestMap(), and timerRequestRobotState().

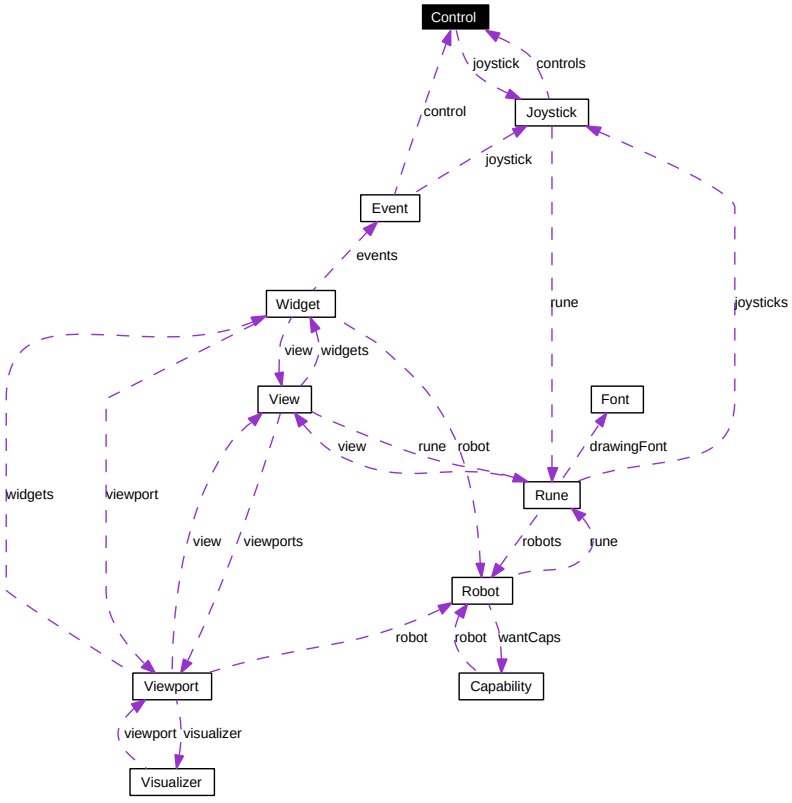
The documentation for this struct was generated from the following file:

- **rune.h**

A.3 Control Struct Reference

`#include <rune.h>`

Collaboration diagram for Control:



Data Fields

- **Joystick * joystick**
- **ControlType type**
- char **name** [256]

- int **index**
- bool **calibrate**
- int **min**
- int **max**
- bool **invert**

A.3.1 Detailed Description

Defines the configuration for a single control of a logical joystick.

This single unified structure handles all control types: axes, balls, buttons, and hat switches.

Definition at line 297 of file rune.h.

A.3.2 Field Documentation

A.3.2.1 bool Control::calibrate

Whether or not this Control should be manually calibrated. True if calibration is required. Applies to axis-type controls only.

Definition at line 319 of file rune.h.

Referenced by parseControl().

A.3.2.2 int Control::index

The integer index of this Control; this maps onto the type-appropriate control array of the logical joystick.

Definition at line 314 of file rune.h.

Referenced by parseControl(), and parseEvent().

A.3.2.3 bool Control::invert

Whether or not this Control's direction should be inverted. Applies to axis-type controls only.

Definition at line 334 of file rune.h.

Referenced by parseControl(), widgetSetImageRequest(), and widgetSetSpeed().

A.3.2.4 Joystick* Control::joystick

A pointer to the Control's parent **Joystick**(p. 35).

Definition at line 301 of file rune.h.

Referenced by initJoysticks(), and parseJoystick().

A.3.2.5 int Control::max

The maximum threshold for this Control. Applies to axis-type controls only.

Definition at line 329 of file rune.h.

Referenced by parseControl(), widgetSetImageRequest(), and widgetSetSpeed().

A.3.2.6 int Control::min

The minimum threshold for this Control. Applies to axis-type controls only.

Definition at line 324 of file rune.h.

Referenced by parseControl(), widgetSetImageRequest(), and widgetSetSpeed().

A.3.2.7 char Control::name[256]

The convenience name of this Control, specified in the configuration file.

Definition at line 309 of file rune.h.

Referenced by parseControl(), parseEvent(), and widgetSetSpeed().

A.3.2.8 ControlType Control::type

The enumerated type of this Control.

Definition at line 305 of file rune.h.

Referenced by parseControl(), parseEvent(), and widgetSetImageRequest().

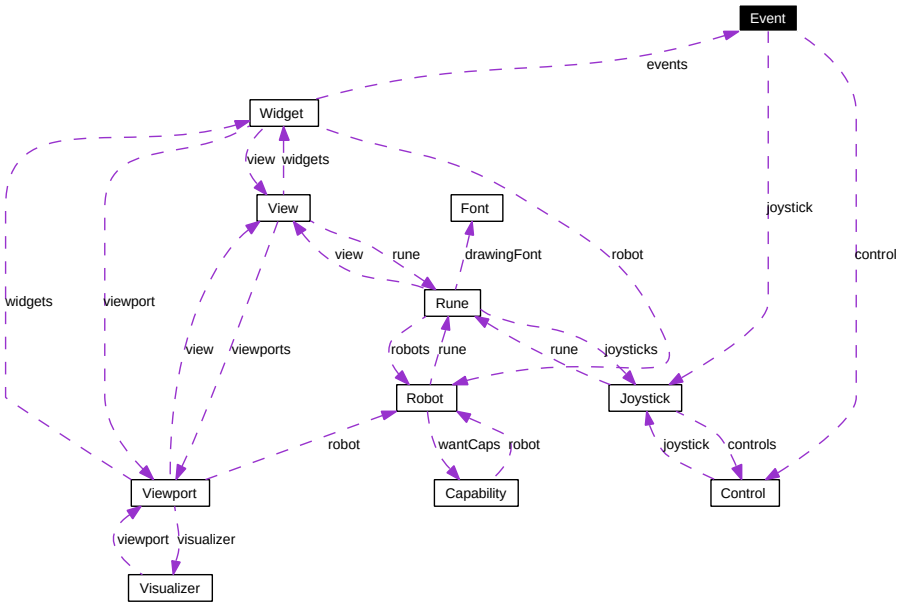
The documentation for this struct was generated from the following file:

- **rune.h**

A.4 Event Struct Reference

`#include <rune.h>`

Collaboration diagram for Event:



Data Fields

- SDL_Event ∗ **event**
- **Joystick** ∗ **joystick**
- **Control** ∗ **control**
- void ∗ **options**

A.4.1 Detailed Description

Associates an SDL event with a **Joystick**(p. 35) and **Control**(p. 26) configuration to determine which events should go to which handlers.

Definition at line 341 of file rune.h.

A.4.2 Field Documentation

A.4.2.1 Control∗ Event::control

The specific joystick control that may be associated with this event.

Definition at line 353 of file rune.h.

Referenced by parseEvent().

A.4.2.2 SDL_Event∗ Event::event

The SDL event whose parameters are used to match events.

Definition at line 345 of file rune.h.

Referenced by parseEvent().

A.4.2.3 Joystick* Event::joystick

The logical joystick configuration that may be associated with this event.

Definition at line 349 of file rune.h.

Referenced by parseEvent().

A.4.2.4 void* Event::options

Arbitrary options associated with this Event.

Definition at line 357 of file rune.h.

Referenced by parseEvent().

The documentation for this struct was generated from the following file:

- **rune.h**

A.5 Font Struct Reference

`#include <rune.h>`

Data Fields

- char **filename** [256]
- int **size**
- TTF_Font * **font**

A.5.1 Detailed Description

Stores a TrueType font and some metadata, including the path to the font file and the desired point size of the font.

Definition at line 364 of file rune.h.

A.5.2 Field Documentation

A.5.2.1 char Font::filename[256]

The path to the TTF file.

Definition at line 368 of file rune.h.

Referenced by runRune().

A.5.2.2 TTF_Font* Font::font

A pointer to the opened font data structure.

Definition at line 376 of file rune.h.

Referenced by runRune().

A.5.2.3 int Font::size

The size in points to use when rendering the font.

Definition at line 372 of file rune.h.

Referenced by runRune().

The documentation for this struct was generated from the following file:

- **rune.h**

A.6 HatSwitchBindings Struct Reference

`#include <rune.h>`

Data Fields

- double **up**
- double **right**
- double **down**
- double **left**

A.6.1 Detailed Description

Defines a set of value bindings for a directional hat switch.

Definition at line 382 of file rune.h.

A.6.2 Field Documentation

A.6.2.1 double HatSwitchBindings::down

The value reported when the hat switch is pressed down.

Definition at line 394 of file rune.h.

A.6.2.2 double HatSwitchBindings::left

The value reported when the hat switch is pressed left.

Definition at line 398 of file rune.h.

A.6.2.3 double HatSwitchBindings::right

The value reported when the hat switch is pressed right.

Definition at line 390 of file rune.h.

A.6.2.4 double HatSwitchBindings::up

The value reported when the hat switch is pressed up.

Definition at line 386 of file rune.h.

The documentation for this struct was generated from the following file:

- **rune.h**

A.7 ImageRequest Struct Reference

`#include <rune.h>`

Data Fields

- bool **handled**
- GCM_Image_Request * **request**

A.7.1 Detailed Description

Defines image request data associated with a VIS_VID capability.

Definition at line 404 of file rune.h.

A.7.2 Field Documentation

A.7.2.1 bool ImageRequest::handled

Whether or not this particular image request has been handled.

Definition at line 408 of file rune.h.

Referenced by initCapability(), timerRequestImage(), and visualizeCameraImage().

A.7.2.2 GCM_Image_Request* ImageRequest::request

The image request message that will be sent over IPC.

Definition at line 412 of file rune.h.

Referenced by initCapability(), and timerRequestImage().

The documentation for this struct was generated from the following file:

- **rune.h**

A.8 InterestPoint Struct Reference

`#include <rune.h>`

Data Fields

- GCM_Map_Interest_Type **type**
- double **x**
- double **y**

A.8.1 Detailed Description

Defines an interest point that may be associated with a PRO_MAP capability.

Definition at line 418 of file rune.h.

A.8.2 Field Documentation

A.8.2.1 GCM_Map_Interest_Type InterestPoint::type

The type of the point.

Definition at line 422 of file rune.h.

Referenced by widgetQuit(), widgetSetLandmark(), and widgetSetVictim().

A.8.2.2 double InterestPoint::x

Global x-position of the point in meters.

Definition at line 426 of file rune.h.

Referenced by widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

A.8.2.3 double InterestPoint::y

Global y-position of the point in meters.

Definition at line 430 of file rune.h.

Referenced by widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

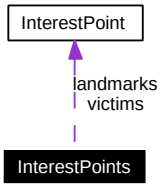
The documentation for this struct was generated from the following file:

- **rune.h**

A.9 InterestPoints Struct Reference

`#include <rune.h>`

Collaboration diagram for InterestPoints:



Data Fields

- **InterestPoint** ∗∗ **landmarks**
- int **nLandmarks**
- **InterestPoint** ∗∗ **victims**
- int **nVictims**

A.9.1 Detailed Description

Defines two resizable arrays for holding victim and landmark data in **InterestPoint**(p. 33) format.

Definition at line 437 of file rune.h.

A.9.2 Field Documentation

A.9.2.1 InterestPoint∗∗ InterestPoints::landmarks

The array of landmark points

Definition at line 441 of file rune.h.

A.9.2.2 int InterestPoints::nLandmarks

The number of landmarks in the array

Definition at line 445 of file rune.h.

Referenced by widgetCorrectLandmark(), and widgetSetLandmark().

A.9.2.3 int InterestPoints::nVictims

The number of victims in the array

Definition at line 453 of file rune.h.

Referenced by widgetSetVictim().

A.9.2.4 InterestPoint** InterestPoints::victims

The array of victim points

Definition at line 449 of file rune.h.

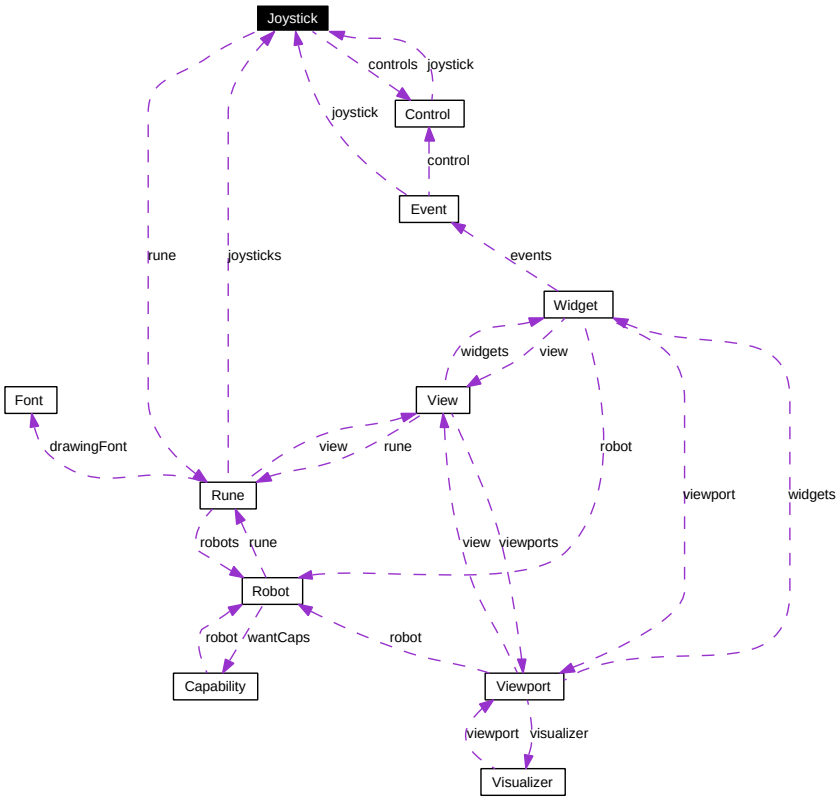
The documentation for this struct was generated from the following file:

- **rune.h**

A.10 Joystick Struct Reference

`#include <rune.h>`

Collaboration diagram for Joystick:



Data Fields

- **Rune** ∗ **rune**
- char **config** [256]
- SDL_Joystick ∗ **joystick**
- **Control** ∗∗ **controls**
- int **nControls**

A.10.1 Detailed Description

Defines the configuration for a logical joystick.

The distinction between a "physical joystick" and a "logical joystick" is as follows: the physical joystick is the one connected to the computer that is used by the human operator; the logical joystick is the internal representation of that connection and configuration.

Definition at line 464 of file rune.h.

A.10.2 Field Documentation

A.10.2.1 char Joystick::config[256]

The name of this joystick configuration.

Definition at line 472 of file rune.h.

Referenced by parseEvent(), and parseJoystick().

A.10.2.2 Control** Joystick::controls

The array of control configurations for this joystick.

Definition at line 480 of file rune.h.

Referenced by parseEvent(), and parseJoystick().

A.10.2.3 SDL_Joystick* Joystick::joystick

A pointer to an open SDL joystick connection.

Definition at line 476 of file rune.h.

Referenced by parseJoystick().

A.10.2.4 int Joystick::nControls

The number of controls configured for this joystick.

Definition at line 484 of file rune.h.

Referenced by parseEvent(), and parseJoystick().

A.10.2.5 Rune* Joystick::rune

A pointer to the Joysticks's parent **Rune**(p. 40).

Definition at line 468 of file rune.h.

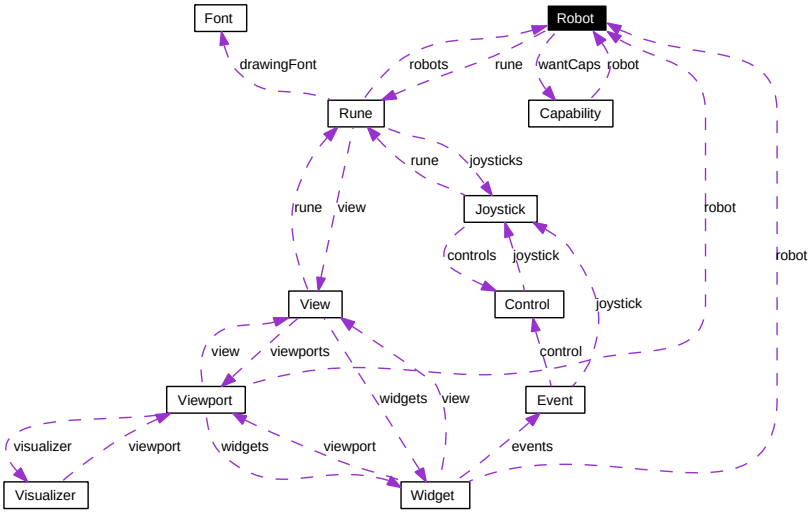
The documentation for this struct was generated from the following file:

- **rune.h**

A.11 Robot Struct Reference

`#include <rune.h>`

Collaboration diagram for Robot:



Data Fields

- **Rune** ∗ **rune**
- char **name** [256]
- char **hostname** [256]
- IPC_CONTEXT_PTR **context**
- **Capability** ∗ **wantCaps** [TOTAL_CAPS]
- GCM_Common_Capabilities **haveCaps**
- GCM_Variable_ModuleInfo **moduleInfo**

A.11.1 Detailed Description

Defines the configuration for an individual robot.

The robot state data, namely the position and orientation, and the camera state, should probably not be a part of this structure.

Definition at line 493 of file rune.h.

A.11.2 Field Documentation

A.11.2.1 IPC_CONTEXT_PTR Robot::context

A pointer to an IPC context identifier that defines the IPC connection associated with this robot.

Definition at line 510 of file rune.h.

Referenced by handleMessageFixed(), handleMessageVariable(), timerRequestImage(), timerRequestMap(), timerRequestRobotState(), timerSendKeepAlive(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

A.11.2.2 GCM_Common_Capabilities Robot::haveCaps

The capabilities available on this robot, as reported by Robomon.

Definition at line 518 of file rune.h.

Referenced by handleMessageCapabilities(), and parseRobot().

A.11.2.3 char Robot::hostname[256]

The robot's network hostname, used for the IPC connection.

Definition at line 505 of file rune.h.

Referenced by parseRobot(), and runRune().

A.11.2.4 GCM_Variable_ModuleInfo Robot::moduleInfo

The modules running on this robot, as reported by Robomon.

Definition at line 522 of file rune.h.

Referenced by parseRobot().

A.11.2.5 char Robot::name[256]

The short name of the robot.

Definition at line 501 of file rune.h.

Referenced by parseRobot().

A.11.2.6 Rune* Robot::rune

A pointer to the Robot's parent **Rune**(p. 40).

Definition at line 497 of file rune.h.

Referenced by handleMessageFixed(), handleMessageVariable(), timerRequestImage(), timerRequestMap(), timerRequestRobotState(), timerSendKeepAlive(), and timerUpdateView().

A.11.2.7 Capability* Robot::wantCaps[TOTAL_CAPS]

The capabilities desired for this robot.

Definition at line 514 of file rune.h.

Referenced by handleMessageCapabilities(), and parseRobot().

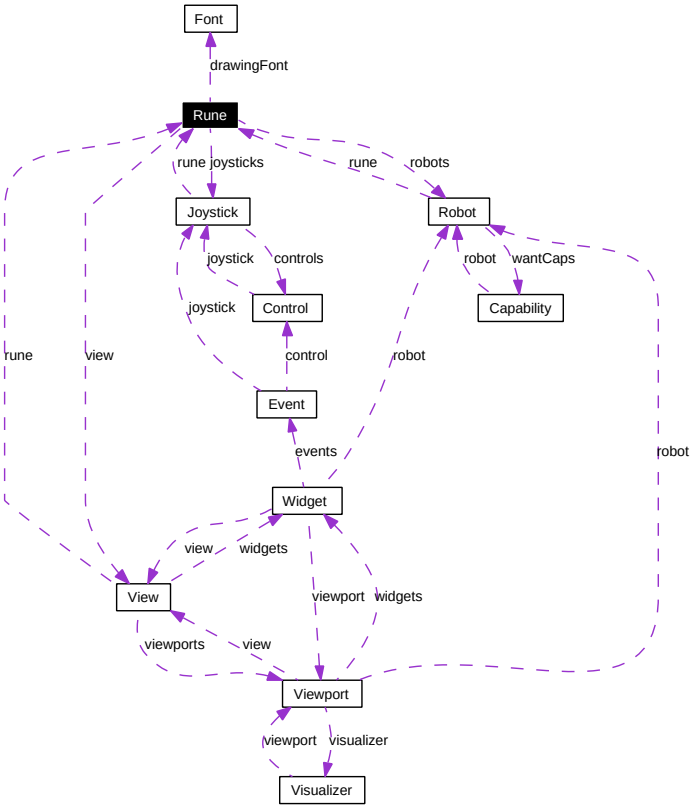
The documentation for this struct was generated from the following file:

- **rune.h**

A.12 Rune Struct Reference

`#include <rune.h>`

Collaboration diagram for Rune:



Data Fields

- GCM_Module_IPC_Data **data**
- GCM_ModuleInfo **info**
- int **runNumber**
- bool **running**
- **Robot** ∗∗ **robots**
- int **nRobots**
- **View** ∗ **view**
- SDL_Surface ∗ **screen**
- **Joystick** ∗∗ **joysticks**
- int **nJoysticks**
- **Font drawingFont**

A.12.1 Detailed Description

Stores module data for an instance of Rune.

Definition at line 528 of file rune.h.

A.12.2 Field Documentation

A.12.2.1 GCM_Module_IPC_Data Rune::data

Module name and process ID.

Definition at line 532 of file rune.h.

Referenced by checkIPC(), handleMessageCommon(), main(), quitRune(), runRune(), and timerSendKeepAlive().

A.12.2.2 Font Rune::drawingFont

The font loaded for on-screen drawing.

Definition at line 572 of file rune.h.

Referenced by runRune().

A.12.2.3 GCM_ModuleInfo Rune::info

Module information, including capabilities and state.

Definition at line 536 of file rune.h.

Referenced by handleMessageCommon(), main(), and runRune().

A.12.2.4 Joystick** Rune::joysticks

The array of configured logical joysticks.

Definition at line 564 of file rune.h.

A.12.2.5 int Rune::nJoysticks

The number of configured logical joysticks.

Definition at line 568 of file rune.h.

A.12.2.6 int Rune::nRobots

The number of robots in use.

Definition at line 552 of file rune.h.

Referenced by runRune().

A.12.2.7 Robot** Rune::robots

The array of **Robot**(p. 37) data structure pointers.

Definition at line 548 of file rune.h.

Referenced by runRune(), and timerSendKeepAlive().

A.12.2.8 bool Rune::running

Whether or not this Rune instance is currently active.

Definition at line 544 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), main(), quitRune(), runRune(), timerRequestImage(), timerRequestMap(), timerRequestRobotState(), timerSendKeepAlive(), and timerUpdateView().

A.12.2.9 int Rune::runNumber

The run number of this Rune instance. Specified in the configuration file.

Definition at line 540 of file rune.h.

A.12.2.10 SDL_Surface* Rune::screen

The SDL surface for the entire screen.

Definition at line 560 of file rune.h.

Referenced by runRune().

A.12.2.11 View* Rune::view

The main **View**(p. 42) data structure.

Definition at line 556 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), and runRune().

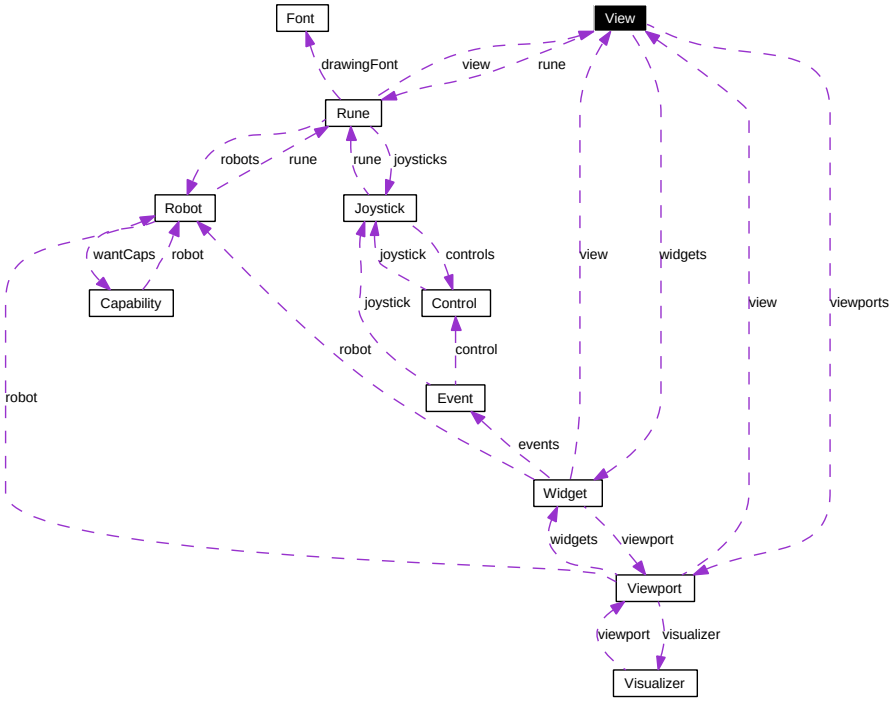
The documentation for this struct was generated from the following file:

- **rune.h**

A.13 View Struct Reference

`#include <rune.h>`

Collaboration diagram for View:



Data Fields

- **Rune** ∗ **rune**
- bool **fullscreen**
- int **xsize**
- int **ysize**
- **Widget** ∗∗ **widgets**
- int **nWidgets**
- **Viewport** ∗∗ **viewports**
- int **nViewports**

A.13.1 Detailed Description

The root of the visual display.

The View has a size that corresponds to its fullscreen or windowed display size. If it is in windowed mode, positioning is handled by the window manager. There can be only one view per application instance at this time.

A View contains some number of Widgets and Viewports. Events handled by View Widgets affect the entire application, as opposed to just their parent **Viewport**(p. 45). Viewports are display areas located within the View's rectangle.

Definition at line 586 of file rune.h.

A.13.2 Field Documentation

A.13.2.1 bool View::fullscreen

Whether the view is in fullscreen or windowed mode. True is fullscreen.

Definition at line 594 of file rune.h.

Referenced by parseView(), and runRune().

A.13.2.2 int View::nViewports

The number of Viewports contained within the View.

Definition at line 620 of file rune.h.

Referenced by parseView(), and runRune().

A.13.2.3 int View::nWidgets

The number of Widgets contained within the View.

Definition at line 612 of file rune.h.

Referenced by parseView().

A.13.2.4 Rune* View::rune

A pointer to the View's parent **Rune**(p. 40).

Definition at line 590 of file rune.h.

Referenced by handleMessageLocal().

A.13.2.5 Viewport** View::viewports

The array of Viewports contained within the View.

Definition at line 616 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), parseView(), and runRune().

A.13.2.6 Widget** View::widgets

The array of Widgets contained within the View.

Definition at line 608 of file rune.h.

Referenced by parseView().

A.13.2.7 int View::xsize

Width of the View in pixels. This cannot exceed the pixel width of the physical screen's current resolution setting.

Definition at line 599 of file rune.h.

Referenced by parseView(), parseViewport(), and runRune().

A.13.2.8 int View::ysize

Height of the View in pixels. This cannot exceed the pixel height of the physical screen's current resolution setting.

Definition at line 604 of file rune.h.

Referenced by parseView(), parseViewport(), and runRune().

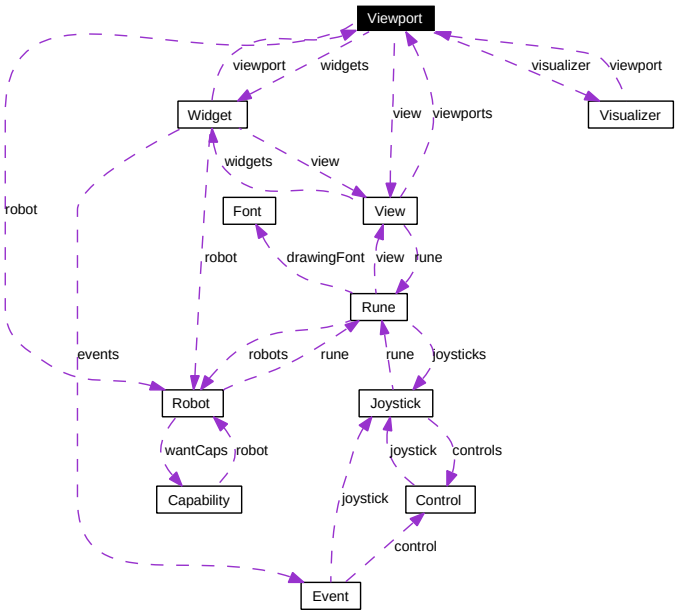
The documentation for this struct was generated from the following file:

- **rune.h**

A.14 Viewport Struct Reference

`#include <rune.h>`

Collaboration diagram for Viewport:



Data Fields

- **View ∗ view**

- bool **visible**
- int **transparency**
- int **xsize**
- int **ysize**
- int **xpos**
- int **ypos**
- int **zpos**
- **Robot** ∗ **robot**
- **Widget** ∗∗ **widgets**
- int **nWidgets**
- **Visualizer** ∗ **visualizer**
- bool **updated**

A.14.1 Detailed Description

An individual data display area.

A Viewport contains some number of Widgets and a **Visualizer**(p. 48). Events handled by the Widgets affect just their parent Viewport. Visualizers convert arbitrary data into image data that can be drawn into the Viewport.

Definition at line 630 of file rune.h.

A.14.2 Field Documentation

A.14.2.1 int Viewport::nWidgets

The number of Widgets contained within the Viewport.

Definition at line 676 of file rune.h.

Referenced by parseViewport().

A.14.2.2 Robot∗ Viewport::robot

A pointer to the robot with which this Viewport is associated.

Definition at line 668 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), parseViewport(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

A.14.2.3 int Viewport::transparency

Alpha channel value, from 0 to 255, for the entire Viewport.

Definition at line 642 of file rune.h.

Referenced by parseViewport().

A.14.2.4 bool Viewport::updated

Whether or not newly Visualized data is available for drawing.

Definition at line 684 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), and handleMessageVariable().

A.14.2.5 View* Viewport::view

A pointer to the Viewport's parent **View**(p. 42).

Definition at line 634 of file rune.h.

Referenced by parseView(), and parseViewport().

A.14.2.6 bool Viewport::visible

Whether or not the Viewport is currently visible. True if visible.

Definition at line 638 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), and parseViewport().

A.14.2.7 Visualizer* Viewport::visualizer

The Viewport's **Visualizer**(p. 48), which renders data for display.

Definition at line 680 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), parseViewport(), and updateViewport().

A.14.2.8 Widget** Viewport::widgets

The array of Widgets contained within the Viewport.

Definition at line 672 of file rune.h.

Referenced by parseViewport().

A.14.2.9 int Viewport::xpos

Horizontal position in pixels relative to left side of the parent **View**(p. 42).

Definition at line 656 of file rune.h.

Referenced by parseViewport(), widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

A.14.2.10 int Viewport::xsize

Width of the Viewport in pixels. This cannot exceed the pixel width of the parent **View**(p. 42).

Definition at line 647 of file rune.h.

Referenced by parseViewport(), widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

A.14.2.11 int Viewport::ypos

Vertical position in pixels relative to top of the parent **View**(p. 42).

Definition at line 660 of file rune.h.

Referenced by parseViewport(), widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

A.14.2.12 int Viewport::ysize

Height of the Viewport in pixels. This cannot exceed the pixel height of the parent **View**(p. 42).

Definition at line 652 of file rune.h.

Referenced by parseViewport(), widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

A.14.2.13 int Viewport::zpos

The depth layer of the Viewport. Larger values are further in front.

Definition at line 664 of file rune.h.

Referenced by compareViewports(), and parseViewport().

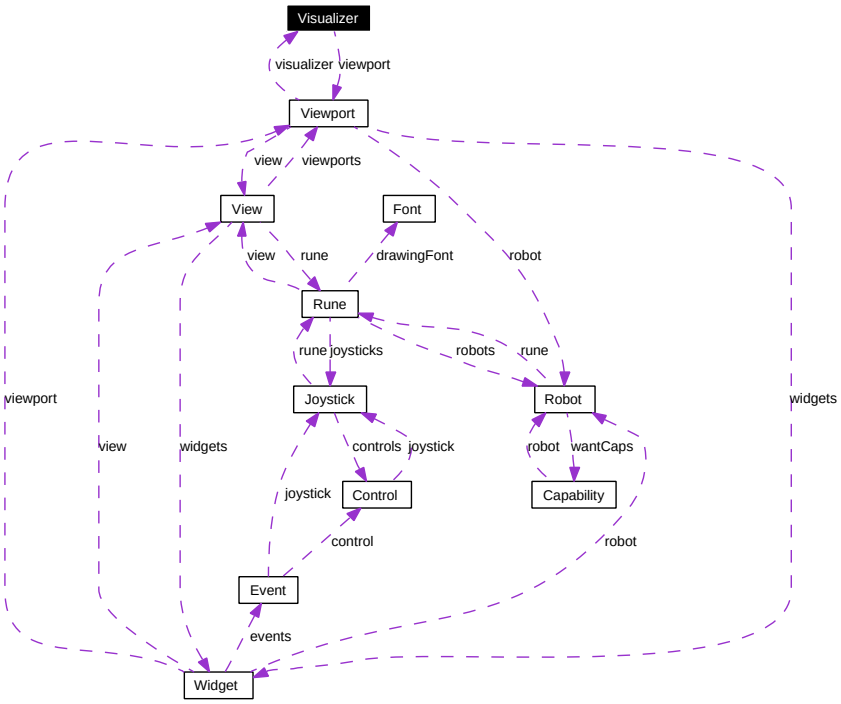
The documentation for this struct was generated from the following file:

- **rune.h**

A.15 Visualizer Struct Reference

`#include <rune.h>`

Collaboration diagram for Visualizer:



Data Fields

- **Viewport** ∗ **viewport**
- int **xsize**
- int **ysize**
- **VisualizerFunction function**
- char **message** [256]
- double **option**
- SDL_Surface ∗ **surface**
- void ∗ **data**

A.15.1 Detailed Description

Converts arbitrary data into visual data that can be drawn into a parent **Viewport**(p. 45).

Definition at line 691 of file rune.h.

A.15.2 Field Documentation

A.15.2.1 void∗ Visualizer::data

A pointer to the arbitrary incoming data, from IPC or **Rune**(p. 40).

Definition at line 724 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), and parseVisualizer().

A.15.2.2 VisualizerFunction Visualizer::function

A pointer to the visualization function that does the actual data conversion work.

Definition at line 708 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), and parseVisualizer().

A.15.2.3 char Visualizer::message[256]

The name of the messages to which this Visualizer listens.

Definition at line 712 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), and handleMessageVariable().

A.15.2.4 double Visualizer::option

An arbitrary numerical value associated with this Visualizer.

Definition at line 716 of file rune.h.

Referenced by parseVisualizer().

A.15.2.5 SDL_Surface* Visualizer::surface

An SDL drawing area for rendering the Visualizer's incoming data.

Definition at line 720 of file rune.h.

Referenced by parseVisualizer(), and updateViewport().

A.15.2.6 Viewport* Visualizer::viewport

A pointer to the Visualizer's parent **Viewport**(p. 45).

Definition at line 695 of file rune.h.

Referenced by handleMessageFixed(), handleMessageLocal(), handleMessageVariable(), parseViewport(), and parseVisualizer().

A.15.2.7 int Visualizer::xsize

Width in pixels of the resulting visual data.

Definition at line 699 of file rune.h.

Referenced by parseVisualizer().

A.15.2.8 int Visualizer::ysize

Height in pixels of the resulting visual data.

Definition at line 703 of file rune.h.

Referenced by parseVisualizer().

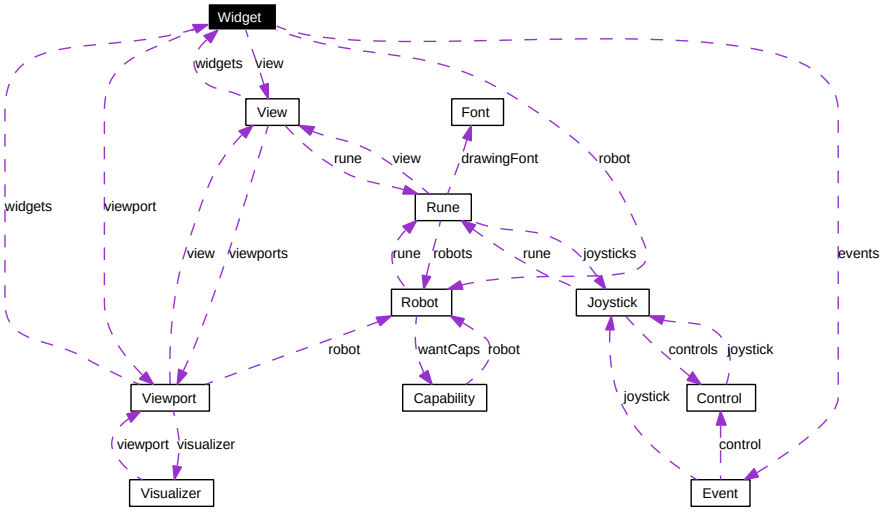
The documentation for this struct was generated from the following file:

- **rune.h**

A.16 Widget Struct Reference

`#include <rune.h>`

Collaboration diagram for Widget:



Data Fields

- **Viewport** ∗ **viewport**
- **View** ∗ **view**
- **WidgetHandler handler**
- **Event** ∗∗ **events**
- int **nEvents**
- **Robot** ∗ **robot**
- void ∗ **history**

A.16.1 Detailed Description

Responds to SDL events.

Definition at line 730 of file rune.h.

A.16.2 Field Documentation

A.16.2.1 Event** Widget::events

An array of events whose parameters are used to match against events that should be handled by this Widget.

Definition at line 747 of file rune.h.

Referenced by parseWidget().

A.16.2.2 WidgetHandler Widget::handler

The pointer to the handler function that does the actual event handling.

Definition at line 742 of file rune.h.

Referenced by parseWidget().

A.16.2.3 void* Widget::history

Arbitrary data stored from the last event handled by this Widget.

Definition at line 760 of file rune.h.

Referenced by parseWidget().

A.16.2.4 int Widget::nEvents

The number of events registered with this Widget.

Definition at line 751 of file rune.h.

Referenced by parseWidget().

A.16.2.5 Robot* Widget::robot

A pointer to the robot with which this Widget is associated.

Todo

Figure out how to not have this pointer here.

Definition at line 756 of file rune.h.

A.16.2.6 View* Widget::view

A pointer to the Widget's parent **View**(p. 42), if any.

Definition at line 738 of file rune.h.

Referenced by parseView(), and parseWidget().

A.16.2.7 Viewport* Widget::viewport

A pointer to the Widget's parent **Viewport**(p. 45), if any.

Definition at line 734 of file rune.h.

Referenced by parseViewport(), and parseWidget().

The documentation for this struct was generated from the following file:

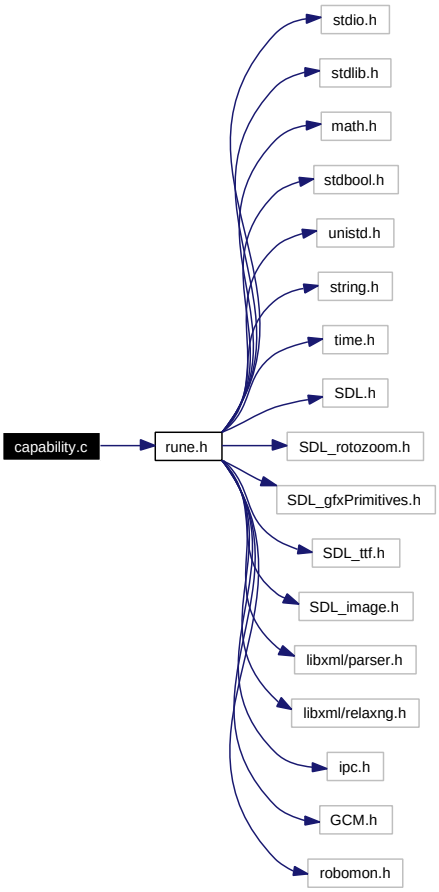
- **rune.h**

B Rune Function Reference

B.1 capability.c File Reference

`#include <rune.h>`

Include dependency graph for capability.c:



Functions

- bool **checkCapabilities** (**Robot** ∗robot)
- void **freeCapability** (**Capability** ∗capability)
- void **freeCapabilities** (**Robot** ∗robot)
- void **freeGCMCapabilities** (GCM_Common_Capabilities ∗caps)
- GCM_Capability **getCapability** (char ∗name)
- char ∗ **getCapabilityName** (GCM_Capability cap)
- void ∗ **getCapabilityQuery** (**Robot** ∗robot, GCM_Capability cap)
- void ∗ **getCapabilityState** (**Robot** ∗robot, GCM_Capability cap)
- char ∗ **getCapabilityString** (GCM_Capability cap)
- GCM_ModuleInfo ∗ **getModuleWithCapability** (**Robot** ∗robot, GCM_Capability cap)
- void **initCapability** (**Capability** ∗capability)

B.1.1 Detailed Description

Contains functions for initializing and managing GCM_Capability related data structures and the robot software modules with which they are associated.

Author:
Nicolas Ward '05

Date:
2005.03.30

Definition in file **capability.c**.

B.1.2 Function Documentation

B.1.2.1 bool checkCapabilities (Robot * *robot*)

Check if all of the capabilities desired for this robot are actually available and active.

The status of robot capabilities is reported by Robomon, handled elsewhere, and stored in the **Robot**(p. 37) data structure.

Parameters:
← ***robot*** A pointer to this robot's **Robot**(p. 37) data structure.

Returns:
True if all modules on the given robot are ready, false otherwise.

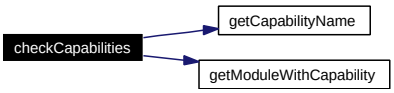
Author:
Nicolas Ward '05

Definition at line 24 of file capability.c.

References getCapabilityName(), and getModuleWithCapability().

Referenced by checkIPC().

Here is the call graph for this function:



B.1.2.2 void freeCapabilities (Robot * *robot*)

Frees an array of **Capability**(p. 23) data structures.

Parameters:
← ***robot*** The **Robot**(p. 37) structure whose **Capability**(p. 23) array is being freed.

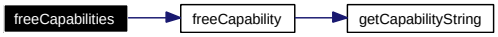
Author:
Nicolas Ward '05

Definition at line 116 of file capability.c.

References freeCapability().

Referenced by freeRobot().

Here is the call graph for this function:



B.1.2.3 void freeCapability (Capability * *capability*)

Frees a **Capability**(p. 23) data structure and all of its children.

Parameters:

← ***capability*** The **Capability**(p. 23) structure being freed.

Author:

Nicolas Ward '05

Todo

Add freeing of capability state.

Todo

Free children of capability query and state properly.

Definition at line 69 of file capability.c.

References getCapabilityString().

Referenced by freeCapabilities().

Here is the call graph for this function:

B.1.2.4 void freeGCMCapabilities (GCM_Common_Capabilities * *caps*)

Free the arrays in a GCM_Common_Capabilities structure.

Parameters:

← ***caps*** A pointer to the structure whose members are to be freed.

Author:

Nicolas Ward '05

Definition at line 139 of file capability.c.

Referenced by freeRobot().

B.1.2.5 GCM_Capability getCapability (char * *name*)

Determines a GCM_Capability enumerated type value based on the equivalent string value.

Parameters:

← ***name*** The string name for the enumerated value.

Returns:

The enumerated capability value.

Author:

Nicolas Ward '05

Definition at line 159 of file capability.c.

Referenced by parseRobot().

B.1.2.6 char* getCapabilityName (GCM_Capability *cap*)

Determines the name of a capability based on the GCM_Capability enumerated type.

Parameters:

← ***cap*** A GCM_Capability enumerated type.

Returns:

The string name of the input capability type.

Author:

Nicolas Ward '05

Definition at line 248 of file capability.c.

Referenced by checkCapabilities(), checkIPC(), getCapabilityQuery(), getCapabilityState(), handleMessageCapabilities(), initCapability(), and parseRobot().

B.1.2.7 void* getCapabilityQuery (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated query data.

Parameters:

← ***robot*** The robot whose capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type whose query variable is desired.

Returns:

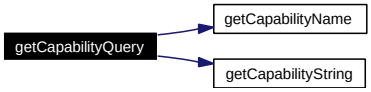
A pointer to the capability's associated query variable.

Definition at line 349 of file capability.c.

References getCapabilityName(), getCapabilityString(), and Capability::query.

Referenced by visualizeCameraImage(), visualizeMapData(), visualizeRangeData(), and widgetSetImageRequest().

Here is the call graph for this function:



B.1.2.8 void* getCapabilityState (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated state data.

Parameters:

← ***robot*** The robot whose capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type whose state variable is desired.

Returns:

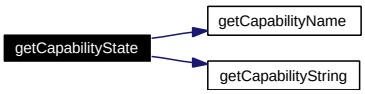
A pointer to the capability's associated state variable.

Definition at line 375 of file capability.c.

References getCapabilityName(), getCapabilityString(), and Capability::state.

Referenced by mapImageToWorld(), mapWorldToImage(), visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizePanData(), visualizeRangeData(), visualizeTiltData(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), and widgetSetVictim().

Here is the call graph for this function:



B.1.2.9 char* getCapabilityString (GCM_Capability *cap*)

Determines a longer description of a capability based on the GCM_Capability enumerated type.

Parameters:

← ***cap*** A GCM_Capability enumerated type.

Returns:
The string description of the input capability type.

Author:
Nicolas Ward '05

Definition at line 400 of file capability.c.

Referenced by checkIPC(), freeCapability(), getCapabilityQuery(), getCapabilityState(), handleMessageCapabilities(), and initCapability().

B.1.2.10 GCM_ModuleInfo* getModuleWithCapability (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated query data.

Parameters:
← ***robot*** The robot whose modules and capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type that will be used to find matching modules.

Returns:
A pointer to the GCM_ModuleInfo structure for the first module found that has the specified capability.

Definition at line 503 of file capability.c.

Referenced by checkCapabilities(), initCapability(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

B.1.2.11 void initCapability (Capability * *capability*)

Initializes a **Capability**(p. 23) data structure based on its GCM_Capability enumerated type.

This function is called by **checkIPC()**(p. 103) after a successful connection to IPC central is made. It expects that all types have been defined properly.

This function handles the initialization of all capabilities that **Rune**(p. 40) knows about. For extensibility, it might be better to break the initialization step into multiple functions, and have this function call those functions as necessary. It might make the code more readable.

Parameters:
← ***capability*** A pointer to the **Capability**(p. 23) being initialized.

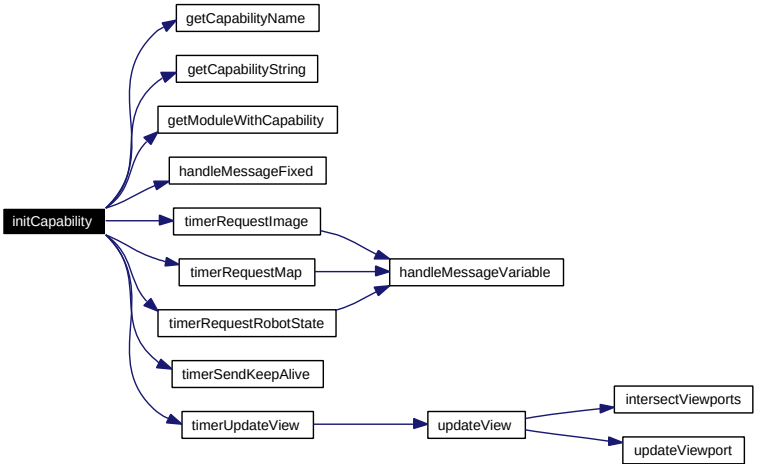
Todo
Maybe break this up into separate initializer functions?

Definition at line 531 of file capability.c.

References getCapabilityName(), getCapabilityString(), getModuleWithCapability(), ImageRequest::handled, CommonRequest::handled, handleMessageFixed(), Capability::nTimers, Capability::query, R_ALIVE_INTERVAL, R_IMAGE_INTERVAL, R_MAP_INTERVAL, R_NAV_INTERVAL, ImageRequest::request, CommonRequest::request, Capability::state, timerRequestImage(), timerRequestMap(), timerRequestRobotState(), Capability::timers, timerSendKeepAlive(), and timerUpdateView().

Referenced by checkIPC().

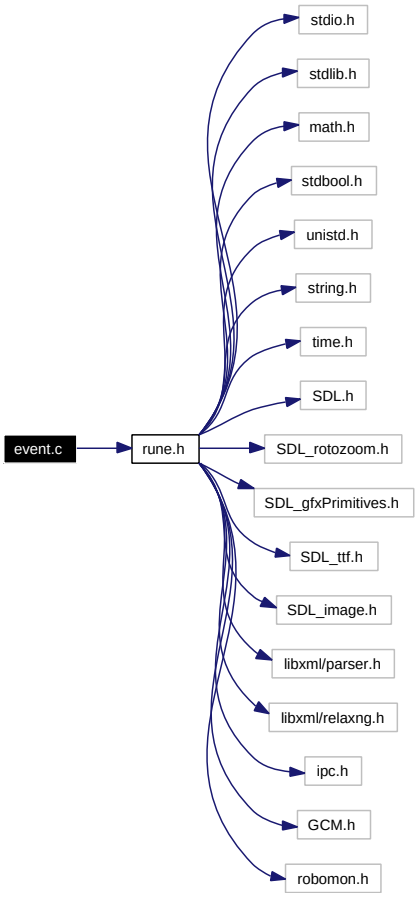
Here is the call graph for this function:



B.2 event.c File Reference

`#include <rune.h>`

Include dependency graph for event.c:



Functions

- bool **compareEvents** (**Event** ∗event, SDL_Event ∗sdlEvent)
- char ∗ **getEventTypeString** (int type)
- void **handleEvent** (SDL_Event ∗event, **Rune** ∗rune)

B.2.1 Detailed Description

Contains event handling and processing functions.

Author:
Nicolas Ward ’05

Date:
2005.03.19

Definition in file **event.c**.

B.2.2 Function Documentation

B.2.2.1 bool compareEvents (Event * *event*, SDL_Event * *sdlEvent*)

Compares the parameters of an SDL_Event that occurred with an **Event**(p. 28) configured as part of a **Widget**(p. 51).

Parameters:

← ***event*** The reference **Event**(p. 28).

← ***sdlEvent*** The new SDL_event whose parameters are being checked.

Returns:

True if the input events are equivalent, false otherwise.

Author:

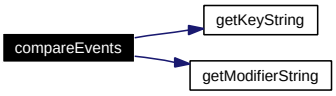
Nicolas Ward '05

Definition at line 21 of file event.c.

References getKeyString(), and getModifierString().

Referenced by handleEvent().

Here is the call graph for this function:



B.2.2.2 char* getEventTypeString (int *type*)

Determines the string representation of an SDL_Event type.

Parameters:

← ***type*** The integer event type.

Returns:

The string name of that event type.

Author:

Nicolas Ward '05

Definition at line 138 of file event.c.

Referenced by handleEvent(), and parseEvent().

B.2.2.3 void handleEvent (SDL_Event * *event*, Rune * *rune*)

Processes SDL events and passes them off to the appropriate event handler.

Parameters:

← ***event*** The SDL event that triggered the handler.

← ***rune*** The **Rune**(p. 40) data structure.

Author:

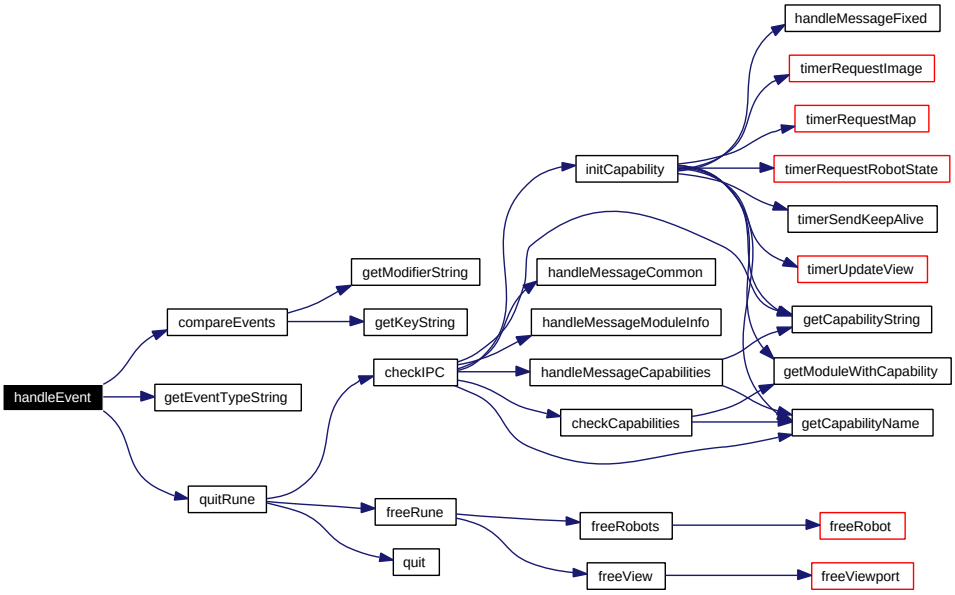
Nicolas Ward '05

Definition at line 192 of file event.c.

References compareEvents(), getEventTypeString(), and quitRune().

Referenced by main(), and runRune().

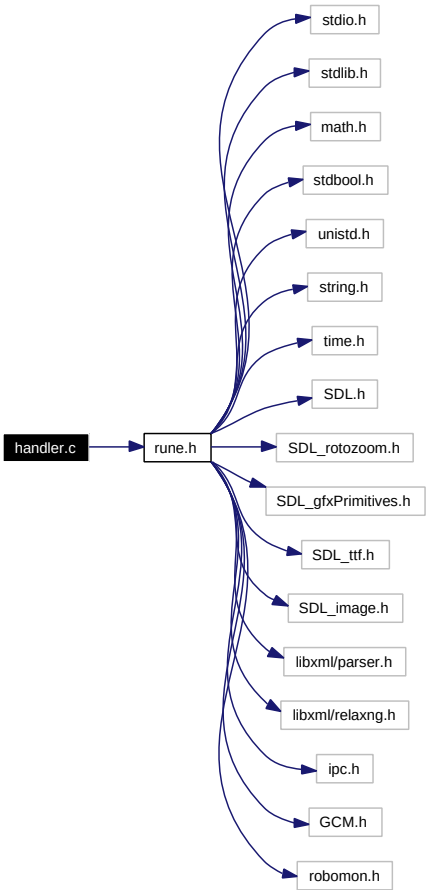
Here is the call graph for this function:



B.3 handler.c File Reference

`#include <rune.h>`

Include dependency graph for handler.c:



Functions

- void **handleMessageCapabilities** (MSG_INSTANCE msgInstance, void ∗callData, void ∗clientData)
- void **handleMessageCommon** (MSG_INSTANCE msgInstance, void ∗callData, void ∗clientData)
- void **handleMessageFixed** (MSG_INSTANCE msgInstance, void ∗callData, void ∗client-Data)
- void **handleMessageLocal** (**Widget** ∗widget, char ∗message, void ∗data)
- void **handleMessageModuleInfo** (MSG_INSTANCE msgInstance, void ∗callData, void ∗clientData)
- void **handleMessageVariable** (MSG_INSTANCE msgInstance, void ∗callData, void ∗clientData)

B.3.1 Detailed Description

Contains functions for handling incoming IPC messages.

Author:
Nicolas Ward ’05

Date:
2005.03.30

Definition in file **handler.c**.

B.3.2 Function Documentation

B.3.2.1 void handleMessageCapabilities (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles Robomon capability listing messages.

Based on capHandler in the rmon-control interface.

Parameters:
← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to a capabilities data structure.

Author:
Nicolas Ward ’05
Fritz Heckel ’05

Todo
Determine if not freeing causes a small memory leak.

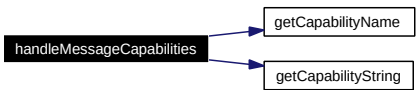
Bug
Freeing the capabilities first causes a segfault.

Definition at line 25 of file handler.c.

References getCapabilityName(), getCapabilityString(), Robot::haveCaps, and Robot::wantCaps.

Referenced by checkIPC().

Here is the call graph for this function:



B.3.2.2 void handleMessageCommon (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles GCM common command messages. If necessary, **Rune**(p. 40) will respond with the appropriate behavior.

Based on commonHandler in the skeleton module.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05
Fritz Heckel '05

Definition at line 100 of file handler.c.

References Rune::data, and Rune::info.

Referenced by checkIPC().

B.3.2.3 void handleMessageFixed (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Processes fixed-length IPC messages from a robot module and passes them off to the appropriate **Visualizer**(p. 48), based on the module's capabilities and the message type.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The incoming fixed-length IPC message received from one of the robot modules.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Definition at line 165 of file handler.c.

References Robot::context, Visualizer::data, Visualizer::function, Visualizer::message, Viewport::robot, Capability::robot, Robot::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by initCapability().

B.3.2.4 void handleMessageLocal (Widget * *widget*, char * *message*, void * *data*)

Passes arbitrary data off to the appropriate **Visualizer**(p. 48), based on the originating **Widget**(p. 51).

Parameters:

← ***widget*** The widget that sent this message

← ***message*** The string name of the message that was sent.

← ***data*** The arbitrary data associated with the message.

Author:

Nicolas Ward '05

Definition at line 238 of file handler.c.

References Visualizer::data, Visualizer::function, Visualizer::message, Viewport::robot, View::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by widgetToggleNightMode().

B.3.2.5 void handleMessageModuleInfo (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles Robomon module information messages.

Based on modHandler in the rmon-control interface.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to a module information data structure.

Author:

Nicolas Ward '05
Fritz Heckel '05

Definition at line 309 of file handler.c.

Referenced by checkIPC().

B.3.2.6 void handleMessageVariable (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Processes variable-length IPC messages from a robot module and passes them off to the appropriate message handler, which should be a **Visualizer**(p. 48).

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The incoming variable-length IPC message received from one of the robot modules.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Definition at line 364 of file handler.c.

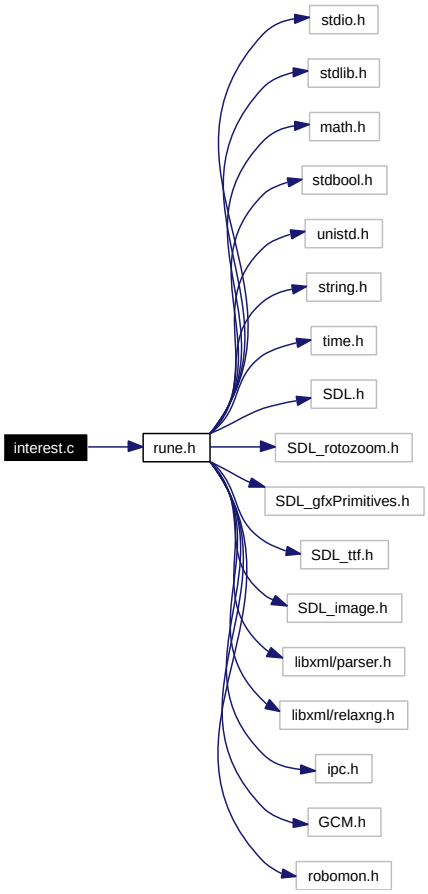
References Robot::context, Visualizer::data, Visualizer::function, Visualizer::message, Viewport::robot, Capability::robot, Robot::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by timerRequestImage(), timerRequestMap(), and timerRequestRobotState().

B.4 interest.c File Reference

`#include <rune.h>`

Include dependency graph for interest.c:



Functions

- void **freeInterestPoints** (**InterestPoints** ∗points)
- **InterestPoint** ∗ **interestLandmarkPeek** (**InterestPoints** ∗points)
- void **interestLandmarkPush** (**InterestPoints** ∗points, **InterestPoint** ∗landmark)
- **InterestPoint** ∗ **interestVictimPeek** (**InterestPoints** ∗points)
- void **interestVictimPush** (**InterestPoints** ∗points, **InterestPoint** ∗victim)

B.4.1 Detailed Description

Containsfunctions which operate on **InterestPoint**(p. 33) and **InterestPoints**(p. 34) data structures.

Author:
Nicolas Ward ’05

Date:
2005.04.29

Definition in file **interest.c**.

B.4.2 Function Documentation

B.4.2.1 void freeInterestPoints (InterestPoints * *points*)

Frees an **InterestPoints**(p. 34) data structure and all of its children.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being freed.

Author:

Nicolas Ward '05

Definition at line 19 of file interest.c.

B.4.2.2 InterestPoint* interestLandmarkPeek (InterestPoints * *points*)

Gets the current landmark.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

Returns:

A pointer to the topmost **InterestPoint**(p. 33).

Author:

Nicolas Ward '05

Definition at line 46 of file interest.c.

Referenced by widgetCorrectLandmark().

B.4.2.3 void interestLandmarkPush (InterestPoints * *points*, InterestPoint * *landmark*)

Adds a new landmark.

Landmarks are used as waypoints in robot navigation, and can be used to correct erroneous robot odometry. They are stored in a push-only stack.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

← ***landmark*** The **InterestPoint**(p. 33) data structure being added.

Author:

Nicolas Ward '05

Definition at line 64 of file interest.c.

Referenced by widgetSetLandmark().

B.4.2.4 InterestPoint* interestVictimPeek (InterestPoints * *points*)

Gets the current victim.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

Returns:

A pointer to the topmost **InterestPoint**(p. 33).

Author:

Nicolas Ward '05

Definition at line 100 of file interest.c.

B.4.2.5 void interestVictimPush (InterestPoints * *points*, InterestPoint * *victim*)

Adds a new victim.

Victims are used as waypoints in robot navigation, and can be used to correct erroneous robot odometry. They are stored in a push-only stack.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

← ***victim*** The **InterestPoint**(p. 33) data structure being added.

Author:

Nicolas Ward '05

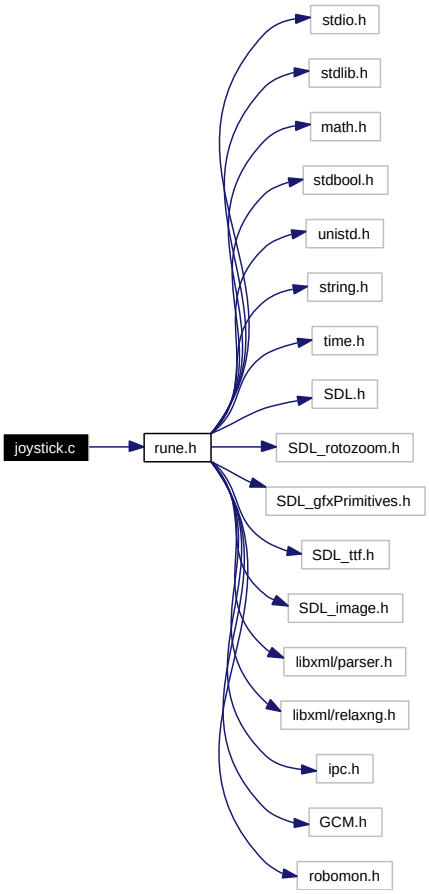
Definition at line 118 of file interest.c.

Referenced by widgetSetVictim().

B.5 joystick.c File Reference

`#include <rune.h>`

Include dependency graph for joystick.c:



Functions

- void **calibrateAxis** (**Joystick** ∗joystick, **Control** ∗control, **Rune** ∗rune)
- int **countControls** (**Joystick** ∗joystick, **ControlType** type)
- **Control** ∗ **getJoystickControl** (**Joystick** ∗joystick, **ControlType** type, int index)
- **ControlType getJoystickControlType** (char ∗name)
- void **initJoysticks** (**Rune** ∗rune)

B.5.1 Detailed Description

Contains functions configuring and connecting to a physical joystick attached to the client computer.

Author:
Nicolas Ward ’05

Date:
2005.03.20

Definition in file **joystick.c**.

B.5.2 Function Documentation

B.5.2.1 void calibrateAxis (Joystick * *joystick*, Control * *control*, Rune * *rune*)

Handles user-controlled calibration of a single joystick axis.

User is prompted to move a specified axis to its maximum, press any button, move the axis to its minimum, and press any button. This ensures that a given axes' extrema are set properly.

Parameters:

← ***joystick*** A pointer to the configuration data structure for the logical joystick being calibrated.

← ***control*** A pointer to the configuration data structure for the axis-type control being calibrated.

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Try to get automatic calibration working.

Definition at line 28 of file joystick.c.

References drawText().

Referenced by initJoysticks().

Here is the call graph for this function:

B.5.2.2 int countControls (Joystick * *joystick*, ControlType *type*)

Counts the number of controls configured for a given type.

Parameters:

← ***joystick*** A pointer to the configuration data structure for the logical joystick whose controls are being counted.

← ***type*** The enumerated control type being counted.

Returns:

The integer count of the specified control type for the specified joystick.

Author:

Nicolas Ward '05

Todo
Figure out why I was counting controls in a weird way.

Definition at line 100 of file joystick.c.

References JOYSTICK_CONTROL_NONE.

Referenced by initJoysticks(), and parseJoystick().

B.5.2.3 Control* getJoystickControl (Joystick * *joystick*, ControlType *type*, int *index*)

Selects a specific control from a joystick configuration.

Since all Controls are stored in a single array, the type and index for that type must be specified.

Parameters:
← ***joystick*** A pointer to the joystick configuration being queried.
← ***type*** The enumerated control type being requested.
← ***index*** The integer index into the logical joystick's control array for the specified type.

Returns:
The matching **Control**(p. 26) configuration data structure.

Author:
Nicolas Ward '05

Todo
Determine if this function is still necessary

Definition at line 138 of file joystick.c.

B.5.2.4 ControlType getJoystickControlType (char * *name*)

Hashes a joystick control type string to an enumerated type.

Parameters:
← ***name*** The string name for the control type being hashed.

Returns:
The enumerated control type.

Author:
Nicolas Ward '05

Definition at line 162 of file joystick.c.

References JOYSTICK_CONTROL_AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, JOYSTICK_CONTROL_HAT_SWITCH, and JOYSTICK_CONTROL_NONE.

Referenced by parseControl().

B.5.2.5 void initJoysticks (Rune * *rune*)

Detects and initializes physical joysticks attached to the client computer.

Parameters:

↔ ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

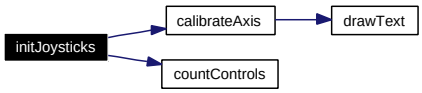
Nicolas Ward '05

Definition at line 189 of file joystick.c.

References calibrateAxis(), countControls(), Control::joystick, JOYSTICK_CONTROL_-AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, and JOYSTICK_-CONTROL_HAT_SWITCH.

Referenced by main(), and runRune().

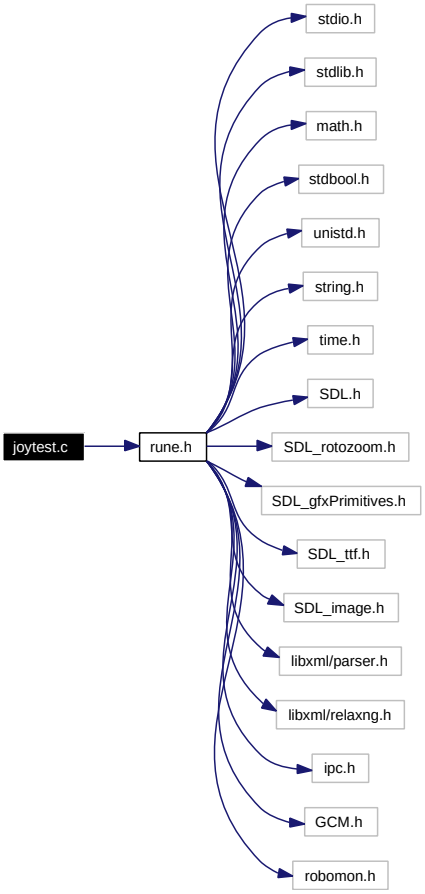
Here is the call graph for this function:



B.6 joytest.c File Reference

`#include <rune.h>`

Include dependency graph for joytest.c:



Functions

- int **main** (int argc, char ∗∗argv)

B.6.1 Detailed Description

Contains a simple main loop to monitor a physical joystick connected to the client computer.

Author:
Nicolas Ward ’05

Date:
2005.03.20

Definition in file **joytest.c**.

B.6.2 Function Documentation

B.6.2.1 int main (int *argc*, char ∗∗ *argv*)

The main joystick querying function.

Checks command line arguments, allocates the **Rune**(p. 40) state data structures, parses the XML configuration file, calls SDL initialization functions, and executes a joystick querying main loop.

Parameters:

← ***argc*** The number of command line arguments.

← ***argv*** The array of command line argument strings.

Returns:

0 on successful execution, -1 on error.

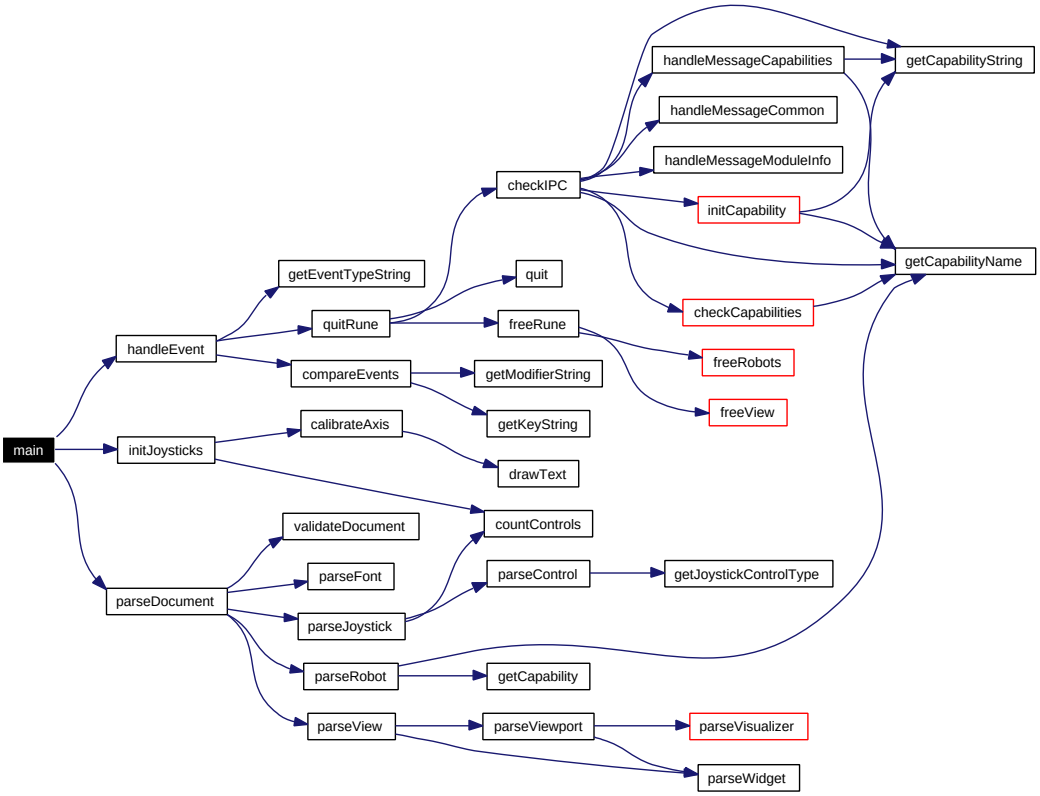
Author:

Nicolas Ward '05

Definition at line 25 of file joytest.c.

References Rune::data, handleEvent(), Rune::info, initJoysticks(), parseDocument(), R_NAME, and Rune::running.

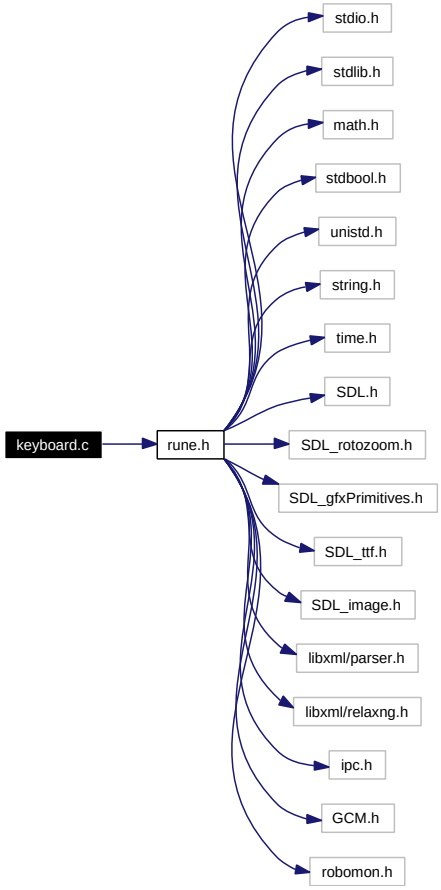
Here is the call graph for this function:



B.7 keyboard.c File Reference

`#include <rune.h>`

Include dependency graph for keyboard.c:



Functions

- int **getButtonState** (char *state)
- SDLKey **getKey** (char *name)
- char * **getKeyString** (SDLKey key)
- SDLMod **getModifier** (char *name)
- char * **getModifierString** (SDLMod mod)

B.7.1 Detailed Description

Contains functions for converting between strings and enumerated types that are associated with keyboard events.

Author:
Nicolas Ward '05

Date:
2005.03.21

Definition in file **keyboard.c**.

B.7.2 Function Documentation

B.7.2.1 int getButtonState (char * *state*)

Deterimes the desired state of a button or key.

Parameters:
← ***state*** The string representation of the button or key's state. This value will be either "pressed" or "released".

Returns:
An enumerated integer value for the state. This value will be either SDL_PRESSED or SDL_-RELEASED.

Author:
Nicolas Ward '05

Definition at line 22 of file keyboard.c.

Referenced by parseEvent().

B.7.2.2 SDLKey getKey (char * *name*)

Determines the enumerated type of a key symbol based on the key name.

Parameters:
← ***name*** The string representation of the key.

Returns:
An enumerated type value for the key.

Author:
Nicolas Ward '05

Definition at line 36 of file keyboard.c.

Referenced by parseEvent().

B.7.2.3 char* getKeyString (SDLKey *key*)

Determines the name of a key based on the enumerated type of the key.

Parameters:
← ***key*** The integer key symbol.

Returns:
The string representation of the key.

Author:
Nicolas Ward '05

Definition at line 427 of file keyboard.c.

Referenced by compareEvents().

B.7.2.4 SDLMod getModifier (char * *name*)

Determines the enumerated type of a key modifier based on the modifier name.

Parameters:
← ***name*** The string representation of the key modifier.

Returns:
An enumerated type value for the key modifier.

Author:
Nicolas Ward '05

Definition at line 832 of file keyboard.c.

Referenced by parseEvent().

B.7.2.5 char* getModifierString (SDLMod *mod*)

Determines the names of key modifiers based on the bitwise ORed enumerated type of the modifier keys.

Parameters:
← ***mod*** An bitwise ORed enumerated type value for the modifier keys.

Returns:
The string representation of the modifier keys.

Author:
Nicolas Ward '05

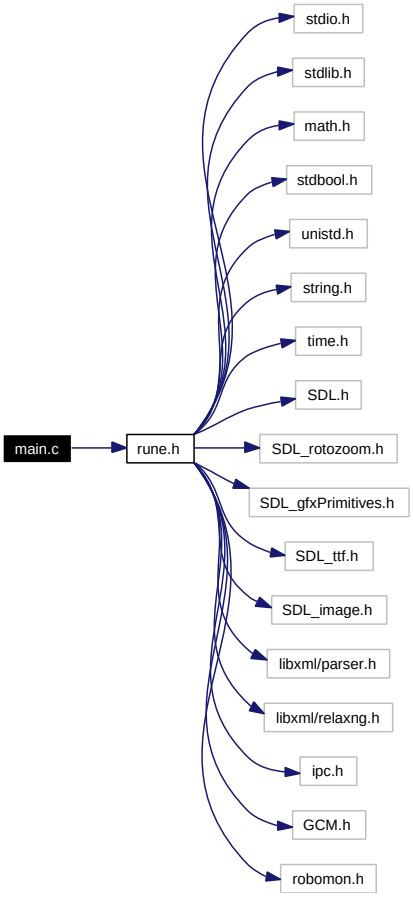
Definition at line 877 of file keyboard.c.

Referenced by compareEvents(), and parseEvent().

B.8 main.c File Reference

`#include <rune.h>`

Include dependency graph for main.c:



Functions

- int **main** (int argc, char ∗∗argv)

B.8.1 Detailed Description

Contains a placeholder main function.

Author:
Nicolas Ward ’05

Date:
2005.03.19

Definition in file **main.c**.

B.8.2 Function Documentation

B.8.2.1 int main (int *argc*, char ** *argv*)

This is a placeholder main function. Calls the **Rune**(p. 40) runtime function.

Parameters:

← ***argc*** The number of command line arguments.

← ***argv*** The array of command line argument strings.

Returns:

The return value of the **Rune**(p. 40) runtime funciton.

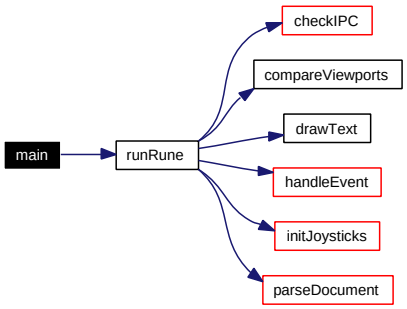
Author:

Nicolas Ward '05

Definition at line 20 of file main.c.

References runRune().

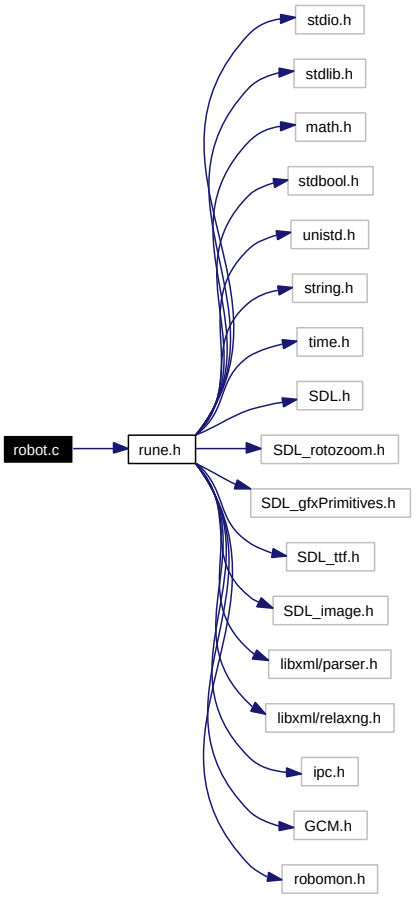
Here is the call graph for this function:



B.9 robot.c File Reference

`#include <rune.h>`

Include dependency graph for robot.c:



Functions

- bool **checkIPC** (**Robot** ∗robot)
- void **freeModuleInfo** (GCM_Variable_ModuleInfo ∗moduleInfo)
- void **freeRobot** (**Robot** ∗robot)
- void **freeRobots** (**Rune** ∗rune)

B.9.1 Detailed Description

Contains functions for configuring and operating on **Robot**(p. 37) data structures.

Author:
Nicolas Ward ’05

Date:
2005.03.21

Definition in file **robot.c**.

B.9.2 Function Documentation

B.9.2.1 bool checkIPC (Robot * *robot*)

Checks if **Rune**(p. 40) is connected to IPC on a particular robot. If it isn't connected, **Rune**(p. 40) will try to connect to IPC on that robot and initialize the connection and other robot data.

Based on smdCheckIPC in the skeleton module.

Parameters:

← ***robot*** A pointer to the **Robot**(p. 37) whose connection is being checked.

Returns:

True if IPC is connected, false otherwise.

Author:

Nicolas Ward '05
Fritz Heckel '05

Todo

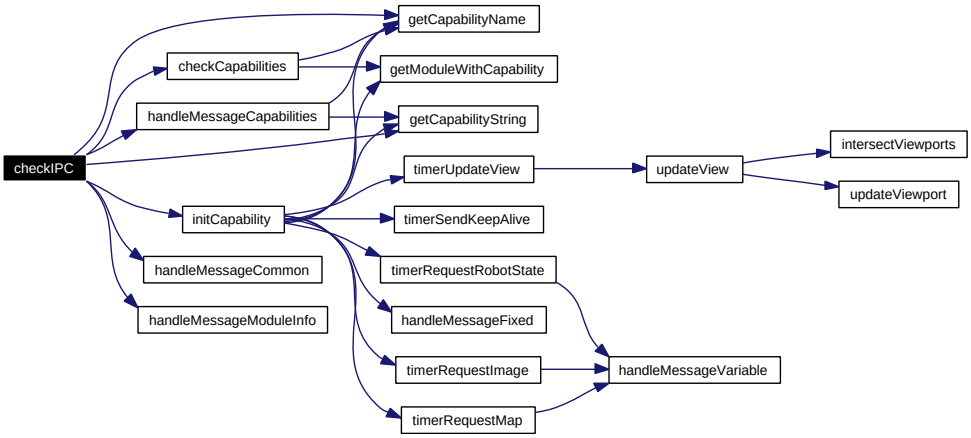
Get the capability request working.

Definition at line 26 of file robot.c.

References checkCapabilities(), Rune::data, getCapabilityName(), getCapabilityString(), handleMessageCapabilities(), handleMessageCommon(), handleMessageModuleInfo(), and initCapability().

Referenced by quitRune(), runRune(), widgetAdjustPan(), widgetAdjustPanTilt(), widgetAdjustTilt(), widgetAdjustZoom(), widgetHomePTZ(), widgetSetSpeed(), and widgetToggleNightMode().

Here is the call graph for this function:



B.9.2.2 void freeModuleInfo (GCM_Variable_ModuleInfo * *moduleInfo*)

Frees the contents of an array of GCM_ModuleInfo structures.

Parameters:

← ***moduleInfo*** A pointer to a GCM_Variable_ModuleInfo structure whose contents is being freed

Author:

Nicolas Ward '05

Todo

Add status printouts.

Definition at line 168 of file robot.c.

Referenced by freeRobot().

B.9.2.3 void freeRobot (Robot * *robot*)

Frees a **Robot**(p. 37) data structure and all of its children.

Parameters:

← ***robot*** The **Robot**(p. 37) structure being freed.

Author:

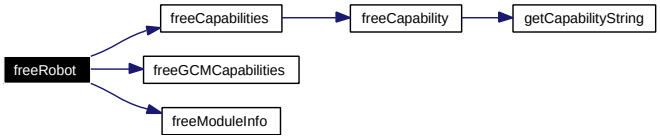
Nicolas Ward '05

Definition at line 187 of file robot.c.

References freeCapabilities(), freeGCMCapabilities(), and freeModuleInfo().

Referenced by freeRobots().

Here is the call graph for this function:



B.9.2.4 void freeRobots (Rune * *rune*)

Frees an array of **Robot**(p. 37) data structures.

Parameters:

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

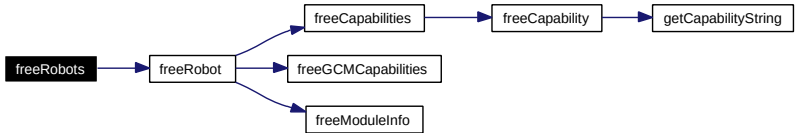
Author:
Nicolas Ward ’05

Definition at line 221 of file robot.c.

References freeRobot().

Referenced by freeRune().

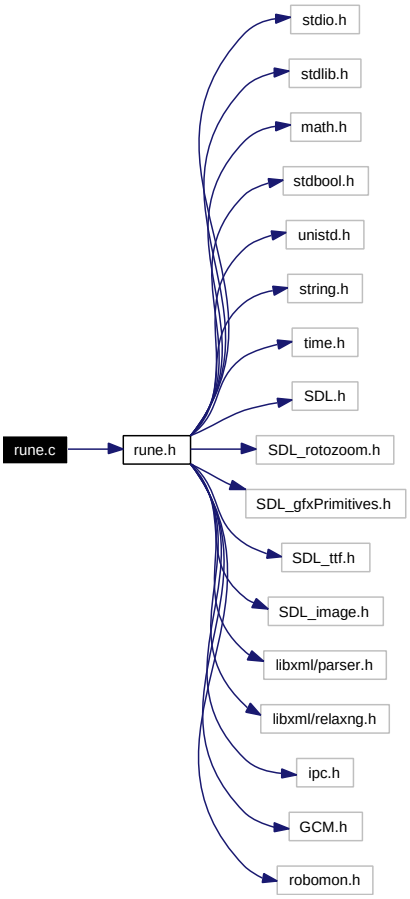
Here is the call graph for this function:



B.10 rune.c File Reference

`#include <rune.h>`

Include dependency graph for rune.c:



Functions

- void **freeRune** (**Rune** ∗rune)
- int **runRune** (int argc, char ∗∗argv)
- void **quitRune** (**Rune** ∗rune)

B.10.1 Detailed Description

Contains the primary runtime functions for **Rune**(p. 40).

Author:
Nicolas Ward ’05

Date:
2005.03.19

Definition in file **rune.c**.

B.10.2 Function Documentation

B.10.2.1 void freeRune (Rune ∗ *rune*)

Frees a **Rune**(p. 40) data structure and all of its children.

Parameters:
← ***rune*** The **Rune**(p. 40) structure being freed.

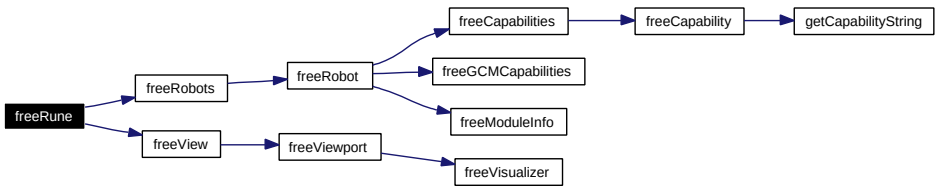
Author:
Nicolas Ward ’05

Definition at line 18 of file rune.c.

References freeRobots(), and freeView().

Referenced by quitRune().

Here is the call graph for this function:



B.10.2.2 void quitRune (Rune * *rune*)

Closes IPC connections, deallocates data structures, and quits **Rune**(p. 40).

Called by the SDL main event loop when a quit event is received.

Parameters:

← ***rune*** A pointer to the **Rune**(p. 40) data strucutre.

Author:

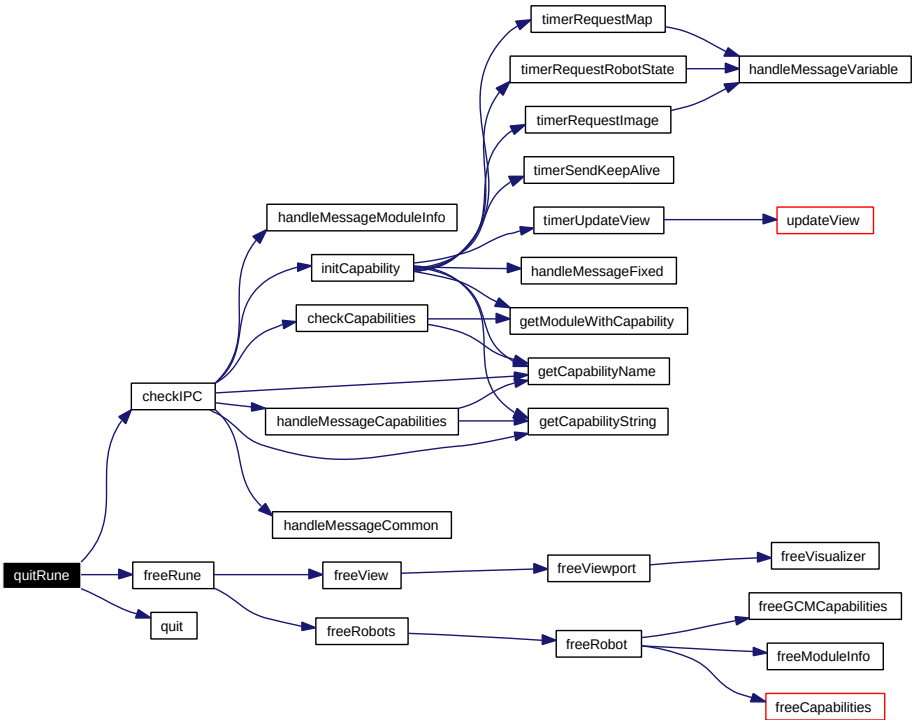
Nicolas Ward '05

Definition at line 261 of file rune.c.

References checkIPC(), Rune::data, freeRune(), quit(), and Rune::running.

Referenced by handleEvent().

Here is the call graph for this function:



B.10.2.3 int runRune (int *argc*, char ** *argv*)

The main runtime function.

Checks command line arguments, allocates the **Rune**(p. 40) state data structures, parses the XML configuration file, calls SDL initialization functions, and executes Rune's main loop.

Parameters:

← ***argc*** The number of command line arguments.

← ***argv*** The array of command line argument strings.

Returns:

0 on successful execution, -1 on error.

Author:

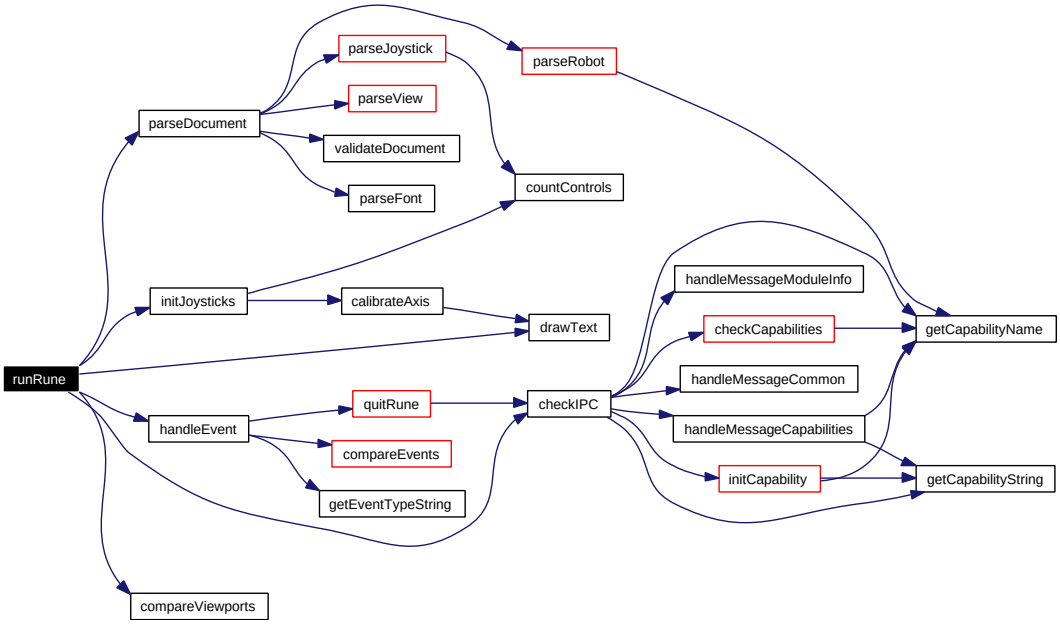
Nicolas Ward '05

Definition at line 53 of file rune.c.

References checkIPC(), compareViewports(), Rune::data, Rune::drawingFont, drawText(), Font::filename, Font::font, View::fullscreen, handleEvent(), Robot::hostname, Rune::info, initJoysticks(), Rune::nRobots, View::nViewports, parseDocument(), R_NAME, R_SDL_INIT_FLAGS, R_SDL_SURFACE_FLAGS, Rune::robots, Rune::running, Rune::screen, Font::size, Rune::view, View::viewports, View::xsize, and View::ysize.

Referenced by main().

Here is the call graph for this function:

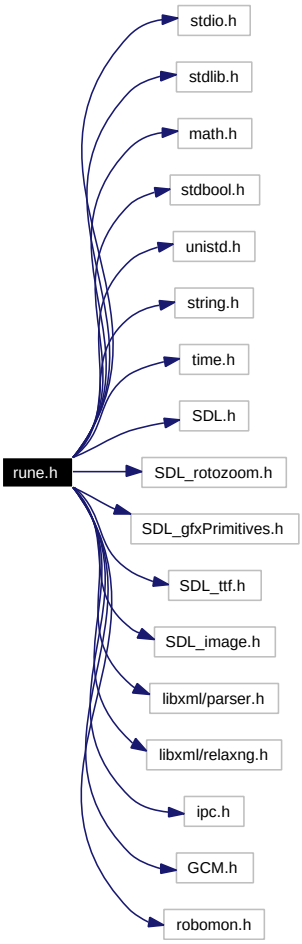


B.11 rune.h File Reference

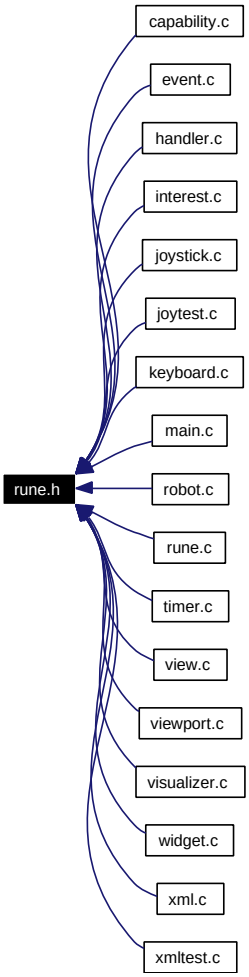
```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <stdbool.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <SDL.h>
#include <SDL_rotozoom.h>
#include <SDL_gfxPrimitives.h>
#include <SDL_ttf.h>
#include <SDL_image.h>
#include <libxml/parser.h>
#include <libxml/relaxng.h>
#include <ipc.h>
#include <GCM.h>
#include <robomon.h>
```

Include dependency graph for rune.h:



This graph shows which files directly or indirectly include this file:



Defines

- #define **R_VERSION** "1.0.0"
- #define **R_DATE** "2005.03.31"
- #define **R_NAME** "Rune Interface"
- #define **R_IMAGE_INTERVAL** 10
- #define **R_NAV_INTERVAL** 10
- #define **R_MAP_INTERVAL** 10000
- #define **R_ALIVE_INTERVAL** 6000
- #define **R_SDL_INIT_FLAGS** SDL_INIT_VIDEO|SDL_INIT_TIMER|SDL_INIT_-JOYSTICK
- #define **R_SDL_SURFACE_FLAGS** SDL_HWSURFACE|SDL_RLEACCEL|SDL_-DOUBLEBUF
- #define **RMASK** 0xff000000
- #define **GMASK** 0x00ff0000
- #define **BMASK** 0x0000ff00

- #define **AMASK** 0x000000ff
- #define **R_TOGGLE_NIGHT_MODE** "RuneToggleNightMode"

Typedefs

- typedef **Capability Capability**
- typedef **CommonRequest CommonRequest**
- typedef **Control Control**
- typedef **Event Event**
- typedef **Font Font**
- typedef **HatSwitchBindings HatSwitchBindings**
- typedef **ImageRequest ImageRequest**
- typedef **InterestPoint InterestPoint**
- typedef **InterestPoints InterestPoints**
- typedef **Joystick Joystick**
- typedef **Robot Robot**
- typedef **Rune Rune**
- typedef **View View**
- typedef **Viewport Viewport**
- typedef **Visualizer Visualizer**
- typedef **Widget Widget**
- typedef void(∗ **VisualizerFunction**)(**Visualizer** ∗visualizer)
- typedef void(∗ **WidgetHandler**)(**Widget** ∗, **Event** ∗, SDL_Event ∗)

Enumerations

- enum **ControlType** {

  JOYSTICK_CONTROL_NONE, **JOYSTICK_CONTROL_AXIS**, **JOYSTICK_-CONTROL_BALL**, **JOYSTICK_CONTROL_BUTTON**,

  JOYSTICK_CONTROL_HAT_SWITCH }

Functions

- bool **checkCapabilities** (**Robot** ∗robot)
- void **freeCapability** (**Capability** ∗capability)
- void **freeCapabilities** (**Robot** ∗robot)
- void **freeGCMCapabilities** (GCM_Common_Capabilities ∗caps)
- GCM_Capability **getCapability** (char ∗name)
- char ∗ **getCapabilityName** (GCM_Capability cap)

- void * **getCapabilityQuery** (**Robot** *robot, GCM_Capability cap)
- void * **getCapabilityState** (**Robot** *robot, GCM_Capability cap)
- char * **getCapabilityString** (GCM_Capability cap)
- GCM_ModuleInfo * **getModuleWithCapability** (**Robot** *robot, GCM_Capability cap)
- void **initCapability** (**Capability** *capability)
- bool **compareEvents** (**Event** *event, SDL_Event *sdlEvent)
- char * **getEventTypeString** (int type)
- void **handleEvent** (SDL_Event *event, **Rune** *rune)
- void **handleMessageCapabilities** (MSG_INSTANCE msgInstance, void *callData, void *clientData)
- void **handleMessageCommon** (MSG_INSTANCE msgInstance, void *callData, void *clientData)
- void **handleMessageFixed** (MSG_INSTANCE msgInstance, void *callData, void *client-Data)
- void **handleMessageLocal** (**Widget** *widget, char *message, void *data)
- void **handleMessageModuleInfo** (MSG_INSTANCE msgInstance, void *callData, void *clientData)
- void **handleMessageVariable** (MSG_INSTANCE msgInstance, void *callData, void *clientData)
- void **freeInterestPoints** (**InterestPoints** *points)
- **InterestPoint** * **interestLandmarkPeek** (**InterestPoints** *points)
- void **interestLandmarkPush** (**InterestPoints** *points, **InterestPoint** *landmark)
- **InterestPoint** * **interestVictimPeek** (**InterestPoints** *points)
- void **interestVictimPush** (**InterestPoints** *points, **InterestPoint** *victim)
- void **calibrateAxis** (**Joystick** *joystick, **Control** *control, **Rune** *rune)
- int **countControls** (**Joystick** *joystick, **ControlType** type)
- **Control** * **getJoystickControl** (**Joystick** *joystick, **ControlType** type, int index)
- **ControlType** **getJoystickControlType** (char *name)
- void **initJoysticks** (**Rune** *rune)
- int **getButtonState** (char *state)
- SDLKey **getKey** (char *name)
- char * **getKeyString** (SDLKey key)
- SDLMod **getModifier** (char *name)
- char * **getModifierString** (SDLMod mod)
- void **freeRune** (**Rune** *rune)
- int **runRune** (int argc, char **argv)
- void **quitRune** (**Rune** *rune)
- bool **checkIPC** (**Robot** *robot)
- void **freeModuleInfo** (GCM_Variable_ModuleInfo *moduleInfo)
- void **freeRobot** (**Robot** *robot)

- void **freeRobots** (**Rune** ∗rune)
- Uint32 **timerRequestImage** (Uint32 interval, void ∗param)
- Uint32 **timerRequestMap** (Uint32 interval, void ∗param)
- Uint32 **timerRequestRobotState** (Uint32 interval, void ∗param)
- Uint32 **timerSendKeepAlive** (Uint32 interval, void ∗param)
- Uint32 **timerUpdateView** (Uint32 interval, void ∗param)
- void **drawText** (SDL_Surface ∗target, TTF_Font ∗font, char ∗text)
- void **freeView** (**View** ∗view)
- void **updateView** (**Rune** ∗rune)
- int **compareViewports** (const void ∗viewportPtr1, const void ∗viewportPtr2)
- void **freeViewport** (**Viewport** ∗viewport)
- bool **intersectViewports** (**Viewport** ∗viewport1, **Viewport** ∗viewport2)
- void **updateViewport** (**Viewport** ∗viewport, **Rune** ∗rune)
- void **drawPTZData** (**Visualizer** ∗visualizer, int shift, int size)
- void **freeVisualizer** (**Visualizer** ∗visualizer)
- void **getVisualizerBindings** (char ∗name, **Visualizer** ∗visualizer)
- void **printImage** (GCM_Common_Image ∗image)
- void **printVisualizer** (**Visualizer** ∗visualizer)
- void **resizeImage** (unsigned char ∗input, int inputW, int inputH, int inputD, unsigned char ∗output, int outputW, int outputH, int outputD)
- void **visualizeCameraImage** (**Visualizer** ∗visualizer)
- void **visualizeGroundPlane** (**Visualizer** ∗visualizer)
- void **visualizeMapData** (**Visualizer** ∗visualizer)
- void **visualizeNightMode** (**Visualizer** ∗visualizer)
- void **visualizePanData** (**Visualizer** ∗visualizer)
- void **visualizeRangeData** (**Visualizer** ∗visualizer)
- void **visualizeTiltData** (**Visualizer** ∗visualizer)
- **WidgetHandler getWidgetHandler** (char ∗name)
- void **mapImageToCamera** (double imageSpaceX, double imageSpaceY, double imageSpaceW, double imageSpaceH, GCM_CameraState ∗camera, double ∗cameraSpaceX, double ∗cameraSpaceY)
- void **mapImageToWorld** (double imageSpaceX, double imageSpaceY, double imageSpaceW, double imageSpaceH, **Robot** ∗robot, double ∗worldSpaceX, double ∗worldSpaceY)
- void **mapCameraToImage** (double cameraSpaceX, double cameraSpaceY, GCM_CameraState ∗camera, double imageSpaceW, double imageSpaceH, double ∗imageSpaceX, double ∗imageSpaceY)
- void **mapWorldToImage** (double worldSpaceX, double worldSpaceY, **Robot** ∗robot, double imageSpaceW, double imageSpaceH, double ∗imageSpaceX, double ∗imageSpaceY)
- void **widgetAdjustPan** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetAdjustPanTilt** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)

- void **widgetAdjustTilt** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetAdjustZoom** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetCorrectLandmark** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetHomePTZ** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetQuit** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetImageRequest** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetLandmark** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetSpeed** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetVictim** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetToggleNightMode** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **parseControl** (**Control** ∗∗controlPtr, xmlDocPtr doc, xmlNodePtr node)
- void **parseDocument** (char ∗filename, **Rune** ∗rune)
- void **parseEvent** (**Event** ∗∗eventPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseFont** (**Font** ∗font, xmlDocPtr doc, xmlNodePtr node)
- void **parseJoystick** (**Joystick** ∗∗joystickPtr, xmlDocPtr doc, xmlNodePtr node)
- void **parseRobot** (**Robot** ∗∗robotPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseView** (**View** ∗∗viewPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseViewport** (**Viewport** ∗∗viewportPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseVisualizer** (**Visualizer** ∗∗visualizerPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseWidget** (**Widget** ∗∗widgetPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **validateDocument** (char ∗filename, xmlDocPtr doc)

B.11.1 Detailed Description

Global header file for **Rune**(p. 40).

All **Rune**(p. 40) files include this file. This file should contain all constant definitions, type definitions, and function declarations.

Author:
Nicolas Ward ’05

Date:
2005.03.19

Definition in file **rune.h**.

B.11.2 Define Documentation

B.11.2.1 #define AMASK 0x000000ff

Definition at line 105 of file rune.h.

Referenced by draw(), main(), visualizeCameraImage(), and visualizeMapData().

B.11.2.2 #define BMASK 0x0000ff00

Definition at line 104 of file rune.h.

Referenced by draw(), drawPTZData(), main(), updateViewport(), visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizeNightMode(), and visualizeRangeData().

B.11.2.3 #define GMASK 0x00ff0000

Definition at line 103 of file rune.h.

Referenced by draw(), drawPTZData(), main(), visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizeNightMode(), and visualizeRangeData().

B.11.2.4 #define R_ALIVE_INTERVAL 6000

Keep-alive delay in ms for the **Rune**(p. 40) IPC connection.

Definition at line 79 of file rune.h.

Referenced by initCapability().

B.11.2.5 #define R_DATE "2005.03.31"

Release date of this version of **Rune**(p. 40).

Definition at line 54 of file rune.h.

B.11.2.6 #define R_IMAGE_INTERVAL 10

Interval in ms when image requests are attempted.

Definition at line 64 of file rune.h.

Referenced by initCapability().

B.11.2.7 #define R_MAP_INTERVAL 10000

Interval in ms when map requests are attempted.

Definition at line 74 of file rune.h.

Referenced by initCapability().

B.11.2.8 #define R_NAME "Rune Interface"

Name of **Rune**(p. 40), used in module data

Definition at line 59 of file rune.h.

Referenced by main(), and runRune().

B.11.2.9 #define R_NAV_INTERVAL 10

Interval in ms when robot state requests are attempted.

Definition at line 69 of file rune.h.

Referenced by initCapability().

B.11.2.10 #define R_SDL_INIT_FLAGS SDL_INIT_VIDEO|SDL_INIT_TIMER|SDL_INIT_JOYSTICK

Initialization flags for SDL. These enable SDL features.

Definition at line 84 of file rune.h.

Referenced by runRune().

B.11.2.11 #define R_SDL_SURFACE_FLAGS SDL_HWSURFACE|SDL_RLEACCEL|SDL_DOUBLEBUF

Surface creations flags for SDL. These enable SDL_Surface features.

Definition at line 92 of file rune.h.

Referenced by drawPTZData(), runRune(), visualizeGroundPlane(), visualizeNightMode(), and visualizeRangeData().

B.11.2.12 #define R_TOGGLE_NIGHT_MODE "RuneToggleNightMode"

Defines a message passed from the widgetToggleNightMode **Widget**(p. 51) handler to the visualizeNightModeIcon **Visualizer**(p. 48) visualization function.

Definition at line 120 of file rune.h.

Referenced by getVisualizerBindings(), and widgetToggleNightMode().

B.11.2.13 #define R_VERSION "1.0.0"

Version of **Rune**(p. 40).

Definition at line 49 of file rune.h.

B.11.2.14 #define RMASK 0xff000000

Definition at line 102 of file rune.h.

Referenced by draw(), drawPTZData(), main(), visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizeNightMode(), and visualizeRangeData().

B.11.3 Typedef Documentation

B.11.3.1 typedef struct Capability Capability

Defines a type for the **Capability**(p. 23) data structure placeholder.

Definition at line 128 of file rune.h.

B.11.3.2 typedef struct CommonRequest CommonRequest

Defines a type for the **CommonRequest**(p. 25) data structure placeholder.

Definition at line 133 of file rune.h.

B.11.3.3 typedef struct Control Control

Defines a type for the **Control**(p. 26) data structure placeholder.

Definition at line 138 of file rune.h.

B.11.3.4 typedef struct Event Event

Defines a type for the **Event**(p. 28) data structure placeholder.

Definition at line 143 of file rune.h.

B.11.3.5 typedef struct Font Font

Defines a type for the **Font**(p. 30) data structure placeholder.

Definition at line 148 of file rune.h.

B.11.3.6 typedef struct HatSwitchBindings HatSwitchBindings

Defines a type for the **HatSwitchBindings**(p. 31) data structure placeholder.

Definition at line 153 of file rune.h.

B.11.3.7 typedef struct ImageRequest ImageRequest

Defines a type for the **ImageRequest**(p. 32) data structure placeholder.

Definition at line 158 of file rune.h.

B.11.3.8 typedef struct InterestPoint InterestPoint

Defines a type for the **InterestPoint**(p. 33) data structure placeholder.

Definition at line 163 of file rune.h.

B.11.3.9 typedef struct InterestPoints InterestPoints

Defines a type for the **InterestPoints**(p. 34) data structure placeholder.

Definition at line 168 of file rune.h.

B.11.3.10 typedef struct Joystick Joystick

Defines a type for the **Joystick**(p. 35) data structure placeholder.

Definition at line 173 of file rune.h.

B.11.3.11 typedef struct Robot Robot

Defines a type for the **Robot**(p. 37) data structure placeholder.

Definition at line 178 of file rune.h.

B.11.3.12 typedef struct Rune Rune

Defines a type for the **Rune**(p. 40) data structure placeholder.

Definition at line 183 of file rune.h.

B.11.3.13 typedef struct View View

Defines a type for the **View**(p. 42) data structure placeholder.

Definition at line 188 of file rune.h.

B.11.3.14 typedef struct Viewport Viewport

Defines a type for the **Viewport**(p. 45) data structure placeholder.

Definition at line 193 of file rune.h.

B.11.3.15 typedef struct Visualizer Visualizer

Defines a type for the **Visualizer**(p. 48) data structure placeholder.

Definition at line 198 of file rune.h.

B.11.3.16 typedef void(∗ VisualizerFunction)(Visualizer ∗visualizer)

All **Visualizer**(p. 48) functions most conform to this function prototype. These functions take the arbitrary data from an IPC message associated with a **Visualizer**(p. 48) data structure and convert them to data that can be drawn onscreen.

Definition at line 214 of file rune.h.

B.11.3.17 typedef struct Widget Widget

Defines a type for the **Widget**(p. 51) data structure placeholder.

Definition at line 203 of file rune.h.

B.11.3.18 typedef void(∗ WidgetHandler)(Widget ∗, Event ∗, SDL_Event ∗)

All **Widget**(p. 51) functions must conform to this function prototype. These functions take the SDL event associated with a **Widget**(p. 51) data structure and its parent **Viewport**(p. 45) and change Rune's state appropriately.

Definition at line 221 of file rune.h.

B.11.4 Enumeration Type Documentation

B.11.4.1 enum ControlType

Enumerated type for logical joystick control configurations.

Enumeration values:

JOYSTICK_CONTROL_NONE

JOYSTICK_CONTROL_AXIS

JOYSTICK_CONTROL_BALL

JOYSTICK_CONTROL_BUTTON

JOYSTICK_CONTROL_HAT_SWITCH

Definition at line 229 of file rune.h.

B.11.5 Function Documentation

B.11.5.1 void calibrateAxis (Joystick ∗ *joystick*, Control ∗ *control*, Rune ∗ *rune*)

Handles user-controlled calibration of a single joystick axis.

User is prompted to move a specified axis to its maximum, press any button, move the axis to its minimum, and press any button. This ensures that a given axes' extrema are set properly.

Parameters:

← ***joystick*** A pointer to the configuration data structure for the logical joystick being calibrated.

← ***control*** A pointer to the configuration data structure for the axis-type control being calibrated.

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Try to get automatic calibration working.

Definition at line 28 of file joystick.c.

References drawText().

Referenced by initJoysticks().

Here is the call graph for this function:

B.11.5.2 bool checkCapabilities (Robot * *robot*)

Check if all of the capabilities desired for this robot are actually available and active.

The status of robot capabilities is reported by Robomon, handled elsewhere, and stored in the **Robot**(p. 37) data structure.

Parameters:

← ***robot*** A pointer to this robot's **Robot**(p. 37) data structure.

Returns:

True if all modules on the given robot are ready, false otherwise.

Author:

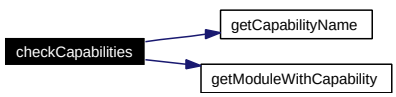
Nicolas Ward '05

Definition at line 24 of file capability.c.

References getCapabilityName(), and getModuleWithCapability().

Referenced by checkIPC().

Here is the call graph for this function:



B.11.5.3 bool checkIPC (Robot * *robot*)

Checks if **Rune**(p. 40) is connected to IPC on a particular robot. If it isn't connected, **Rune**(p. 40) will try to connect to IPC on that robot and initialize the connection and other robot data.

Based on smdCheckIPC in the skeleton module.

Parameters:

← ***robot*** A pointer to the **Robot**(p. 37) whose connection is being checked.

Returns:

True if IPC is connected, false otherwise.

Author:

Nicolas Ward '05
Fritz Heckel '05

Todo

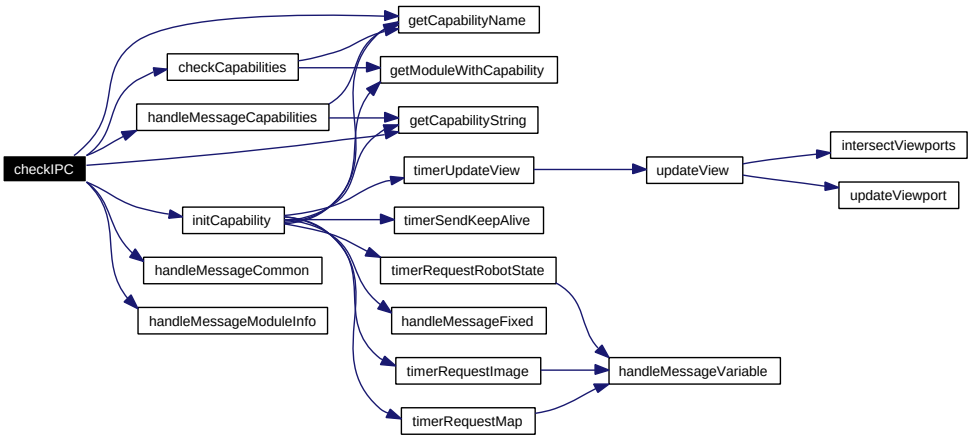
Get the capability request working.

Definition at line 26 of file robot.c.

References checkCapabilities(), Rune::data, getCapabilityName(), getCapabilityString(), handleMessageCapabilities(), handleMessageCommon(), handleMessageModuleInfo(), and initCapability().

Referenced by quitRune(), runRune(), widgetAdjustPan(), widgetAdjustPanTilt(), widgetAdjustTilt(), widgetAdjustZoom(), widgetHomePTZ(), widgetSetSpeed(), and widgetToggleNightMode().

Here is the call graph for this function:



B.11.5.4 bool compareEvents (Event * *event*, SDL_Event * *sdlEvent*)

Compares the parameters of an SDL_Event that occurred with an **Event**(p. 28) configured as part of a **Widget**(p. 51).

Parameters:

← ***event*** The reference **Event**(p. 28).

← ***sdlEvent*** The new SDL_event whose parameters are being checked.

Returns:

True if the input events are equivalent, false otherwise.

Author:

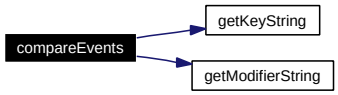
Nicolas Ward '05

Definition at line 21 of file event.c.

References getKeyString(), and getModifierString().

Referenced by handleEvent().

Here is the call graph for this function:



B.11.5.5 int compareViewports (const void * *viewportPtr1*, const void * *viewportPtr2*)

Compares the z position of two Viewports.

Intended for use with qsort(). 0 is returned if the two Viewports have the same zpos; the result is positive if the first is in front of the second, and negative if the second is in front of the first.

Parameters:

← ***viewportPtr1*** A void pointer to the first **Viewport**(p. 45).

← ***viewportPtr2*** A void pointer to the second **Viewport**(p. 45).

Returns:

The difference in the z-position of the input Viewports.

Author:

Nicolas Ward '05

Definition at line 25 of file viewport.c.

References Viewport::zpos.

Referenced by runRune().

B.11.5.6 int countControls (Joystick * *joystick*, ControlType *type*)

Counts the number of controls configured for a given type.

Parameters:

← ***joystick*** A pointer to the configuration data structure for the logical joystick whose controls are being counted.

← ***type*** The enumerated control type being counted.

Returns:

The integer count of the specified control type for the specified joystick.

Author:

Nicolas Ward '05

Todo

Figure out why I was counting controls in a weird way.

Definition at line 100 of file joystick.c.

References JOYSTICK_CONTROL_NONE.

Referenced by initJoysticks(), and parseJoystick().

B.11.5.7 void drawPTZData (Visualizer * *visualizer*, int *shift*, int *size*)

Draws pan and zoom or tilt and zoom data to a **Visualizer**(p. 48) surface.

Parameters:

← ***visualizer*** The pan or tiltVisualizer that called this function.

← ***shift*** The center of the indicator bar, in pixels from one end of the calling Visualizer's surface.

← ***size*** The size of the indicator bar in pixels.

Author:

Nicolas Ward '05

Definition at line 22 of file visualizer.c.

References BMASK, GMASK, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by visualizePanData(), and visualizeTiltData().

B.11.5.8 void drawText (SDL_Surface * *target*, TTF_Font * *font*, char * *text*)

Draws some arbitrary text into a surface.

The text can be drawn in any TTF font. Currently the text is drawn in black on a grey background, which is then blitted into the middle of the target surface.

Parameters:

← ***target*** The surface onto which the text will be drawn.

← ***font*** The font used to render the text.

← ***text*** The string of text to be drawn.

Author:

Nicolas Ward '05

Todo

Add support for arbitrary text coloring.

Definition at line 25 of file view.c.

Referenced by calibrateAxis(), and runRune().

B.11.5.9 void freeCapabilities (Robot * *robot*)

Frees an array of **Capability**(p. 23) data structures.

Parameters:

← ***robot*** The **Robot**(p. 37) structure whose **Capability**(p. 23) array is being freed.

Author:

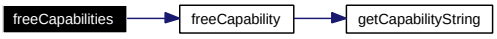
Nicolas Ward '05

Definition at line 116 of file capability.c.

References freeCapability().

Referenced by freeRobot().

Here is the call graph for this function:



B.11.5.10 void freeCapability (Capability * *capability*)

Frees a **Capability**(p. 23) data structure and all of its children.

Parameters:
← ***capability*** The **Capability**(p. 23) structure being freed.

Author:
Nicolas Ward '05

Todo
Add freeing of capability state.

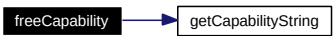
Todo
Free children of capability query and state properly.

Definition at line 69 of file capability.c.

References getCapabilityString().

Referenced by freeCapabilities().

Here is the call graph for this function:



B.11.5.11 void freeGCMCapabilities (GCM_Common_Capabilities * *caps*)

Free the arrays in a GCM_Common_Capabilities structure.

Parameters:
← ***caps*** A pointer to the structure whose members are to be freed.

Author:
Nicolas Ward '05

Definition at line 139 of file capability.c.

Referenced by freeRobot().

B.11.5.12 void freeInterestPoints (InterestPoints * *points*)

Frees an **InterestPoints**(p. 34) data structure and all of its children.

Parameters:
← ***points*** The **InterestPoints**(p. 34) data structure being freed.

Author:
Nicolas Ward '05

Definition at line 19 of file interest.c.

B.11.5.13 void freeModuleInfo (GCM_Variable_ModuleInfo * *moduleInfo*)

Frees the contents of an array of GCM_ModuleInfo structures.

Parameters:

← ***moduleInfo*** A pointer to a GCM_Variable_ModuleInfo structure whose contents is being freed

Author:

Nicolas Ward '05

Todo

Add status printouts.

Definition at line 168 of file robot.c.

Referenced by freeRobot().

B.11.5.14 void freeRobot (Robot * *robot*)

Frees a **Robot**(p. 37) data structure and all of its children.

Parameters:

← ***robot*** The **Robot**(p. 37) structure being freed.

Author:

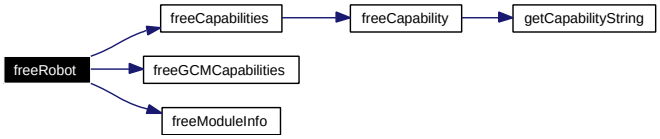
Nicolas Ward '05

Definition at line 187 of file robot.c.

References freeCapabilities(), freeGCMCapabilities(), and freeModuleInfo().

Referenced by freeRobots().

Here is the call graph for this function:



B.11.5.15 void freeRobots (Rune * *rune*)

Frees an array of **Robot**(p. 37) data structures.

Parameters:

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

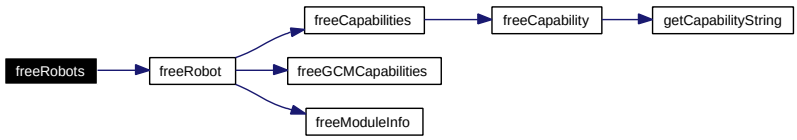
Author:
Nicolas Ward '05

Definition at line 221 of file robot.c.

References freeRobot().

Referenced by freeRune().

Here is the call graph for this function:



B.11.5.16 void freeRune (Rune * *rune*)

Frees a **Rune**(p. 40) data structure and all of its children.

Parameters:
← ***rune*** The **Rune**(p. 40) structure being freed.

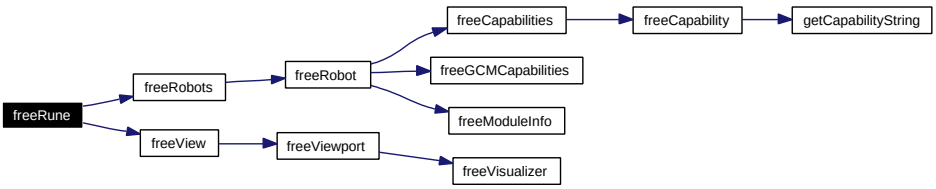
Author:
Nicolas Ward '05

Definition at line 18 of file rune.c.

References freeRobots(), and freeView().

Referenced by quitRune().

Here is the call graph for this function:



B.11.5.17 void freeView (View * *view*)

Frees a **View**(p. 42) data structure and all of its children.

Parameters:
← ***view*** The **View**(p. 42) structure being freed.

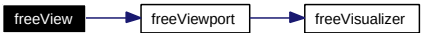
Author:
Nicolas Ward '05

Definition at line 94 of file view.c.

References freeViewport().

Referenced by freeRune().

Here is the call graph for this function:



B.11.5.18 void freeViewport (Viewport * *viewport*)

Frees a **Viewport**(p. 45) data structure and all of its children

Parameters:
← ***viewport*** The **Viewport**(p. 45) structure to be freed.

Author:
Nicolas Ward '05

Definition at line 42 of file viewport.c.

References freeVisualizer().

Referenced by freeView().

Here is the call graph for this function:

B.11.5.19 void freeVisualizer (Visualizer * *visualizer*)

Frees a **Visualizer**(p. 48) data structure.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) structure to be freed.

Author:
Nicolas Ward '05

Definition at line 131 of file visualizer.c.

Referenced by freeViewport().

B.11.5.20 int getButtonState (char * *state*)

Deterimes the desired state of a button or key.

Parameters:

← ***state*** The string representation of the button or key's state. This value will be either "pressed" or "released".

Returns:

An enumerated integer value for the state. This value will be either SDL_PRESSED or SDL_-RELEASED.

Author:

Nicolas Ward '05

Definition at line 22 of file keyboard.c.

Referenced by parseEvent().

B.11.5.21 GCM_Capability getCapability (char * *name*)

Determines a GCM_Capability enumerated type value based on the equivalent string value.

Parameters:

← ***name*** The string name for the enumerated value.

Returns:

The enumerated capability value.

Author:

Nicolas Ward '05

Definition at line 159 of file capability.c.

Referenced by parseRobot().

B.11.5.22 char* getCapabilityName (GCM_Capability *cap*)

Determines the name of a capability based on the GCM_Capability enumerated type.

Parameters:

← ***cap*** A GCM_Capability enumerated type.

Returns:

The string name of the input capability type.

Author:
Nicolas Ward '05

Definition at line 248 of file capability.c.

Referenced by checkCapabilities(), checkIPC(), getCapabilityQuery(), getCapabilityState(), handleMessageCapabilities(), initCapability(), and parseRobot().

B.11.5.23 void* getCapabilityQuery (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated query data.

Parameters:
← ***robot*** The robot whose capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type whose query variable is desired.

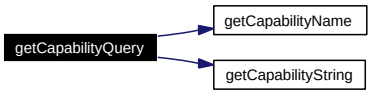
Returns:
A pointer to the capability's associated query variable.

Definition at line 349 of file capability.c.

References getCapabilityName(), getCapabilityString(), and Capability::query.

Referenced by visualizeCameraImage(), visualizeMapData(), visualizeRangeData(), and widgetSetImageRequest().

Here is the call graph for this function:



B.11.5.24 void* getCapabilityState (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated state data.

Parameters:
← ***robot*** The robot whose capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type whose state variable is desired.

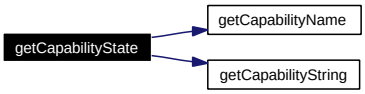
Returns:
A pointer to the capability's associated state variable.

Definition at line 375 of file capability.c.

References getCapabilityName(), getCapabilityString(), and Capability::state.

Referenced by mapImageToWorld(), mapWorldToImage(), visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizePanData(), visualizeRangeData(), visualizeTiltData(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), and widgetSetVictim().

Here is the call graph for this function:



B.11.5.25 char* getCapabilityString (GCM_Capability *cap*)

Determines a longer description of a capability based on the GCM_Capability enumerated type.

Parameters:

← ***cap*** A GCM_Capability enumerated type.

Returns:

The string description of the input capability type.

Author:

Nicolas Ward '05

Definition at line 400 of file capability.c.

Referenced by checkIPC(), freeCapability(), getCapabilityQuery(), getCapabilityState(), handleMessageCapabilities(), and initCapability().

B.11.5.26 char* getEventTypeString (int *type*)

Determines the string representation of an SDL_Event type.

Parameters:

← ***type*** The integer event type.

Returns:

The string name of that event type.

Author:

Nicolas Ward '05

Definition at line 138 of file event.c.

Referenced by handleEvent(), and parseEvent().

B.11.5.27 Control* getJoystickControl (Joystick * *joystick*, ControlType *type*, int *index*)

Selects a specific control from a joystick configuration.

Since all Controls are stored in a single array, the type and index for that type must be specified.

Parameters:

← ***joystick*** A pointer to the joystick configuration being queried.

← ***type*** The enumerated control type being requested.

← ***index*** The integer index into the logical joystick's control array for the specified type.

Returns:

The matching **Control**(p. 26) configuration data structure.

Author:

Nicolas Ward '05

Todo

Determine if this function is still necessary

Definition at line 138 of file joystick.c.

B.11.5.28 ControlType getJoystickControlType (char * *name*)

Hashes a joystick control type string to an enumerated type.

Parameters:

← ***name*** The string name for the control type being hashed.

Returns:

The enumerated control type.

Author:

Nicolas Ward '05

Definition at line 162 of file joystick.c.

References JOYSTICK_CONTROL_AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, JOYSTICK_CONTROL_HAT_SWITCH, and JOYSTICK_CONTROL_NONE.

Referenced by parseControl().

B.11.5.29 SDLKey getKey (char * *name*)

Determines the enumerated type of a key symbol based on the key name.

Parameters:
← ***name*** The string representation of the key.

Returns:
An enumerated type value for the key.

Author:
Nicolas Ward '05

Definition at line 36 of file keyboard.c.

Referenced by parseEvent().

B.11.5.30 char* getKeyString (SDLKey *key*)

Determines the name of a key based on the enumerated type of the key.

Parameters:
← ***key*** The integer key symbol.

Returns:
The string representation of the key.

Author:
Nicolas Ward '05

Definition at line 427 of file keyboard.c.

Referenced by compareEvents().

B.11.5.31 SDLMod getModifier (char * *name*)

Determines the enumerated type of a key modifier based on the modifier name.

Parameters:
← ***name*** The string representation of the key modifier.

Returns:
An enumerated type value for the key modifier.

Author:
Nicolas Ward '05

Definition at line 832 of file keyboard.c.

Referenced by parseEvent().

B.11.5.32 char* getModifierString (SDLMod *mod*)

Determines the names of key modifiers based on the bitwise ORed enumerated type of the modifier keys.

Parameters:

← ***mod*** An bitwise ORed enumerated type value for the modifier keys.

Returns:

The string representation of the modifier keys.

Author:

Nicolas Ward '05

Definition at line 877 of file keyboard.c.

Referenced by compareEvents(), and parseEvent().

B.11.5.33 GCM_ModuleInfo* getModuleWithCapability (Robot * *robot*, GCM_Capability *cap*)

Checks if the specified capability was configured, and then returns its associated query data.

Parameters:

← ***robot*** The robot whose modules and capabilities are being searched.

← ***cap*** A GCM_Capability enumerated type that will be used to find matching modules.

Returns:

A pointer to the GCM_ModuleInfo structure for the first module found that has the specified capability.

Definition at line 503 of file capability.c.

Referenced by checkCapabilities(), initCapability(), widgetCorrectLandmark(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

B.11.5.34 void getVisualizerBindings (char * *name*, Visualizer * *visualizer*)

Hashes a **Visualizer**(p. 48) function name to a VisualizerFunction pointer and a string name of an IPC message.

Parameters:

← ***name*** The string name of the function being hashed.

↔ ***visualizer*** The **Visualizer**(p. 48) that will be bound to the returned visualization function and data message.

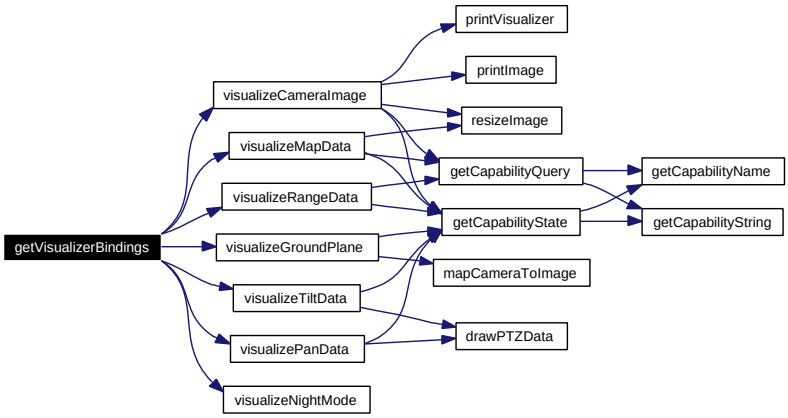
Author:
Nicolas Ward '05

Definition at line 159 of file visualizer.c.

References R_TOGGLE_NIGHT_MODE, visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizeNightMode(), visualizePanData(), visualizeRangeData(), and visualizeTiltData().

Referenced by parseVisualizer().

Here is the call graph for this function:



B.11.5.35 WidgetHandler getWidgetHandler (char * *name*)

Hashes a **Widget**(p. 51) handler function name to a WidgetHandler pointer.

Parameters:
← ***name*** The string name of the function being hashed.

Returns:
The function pointer.

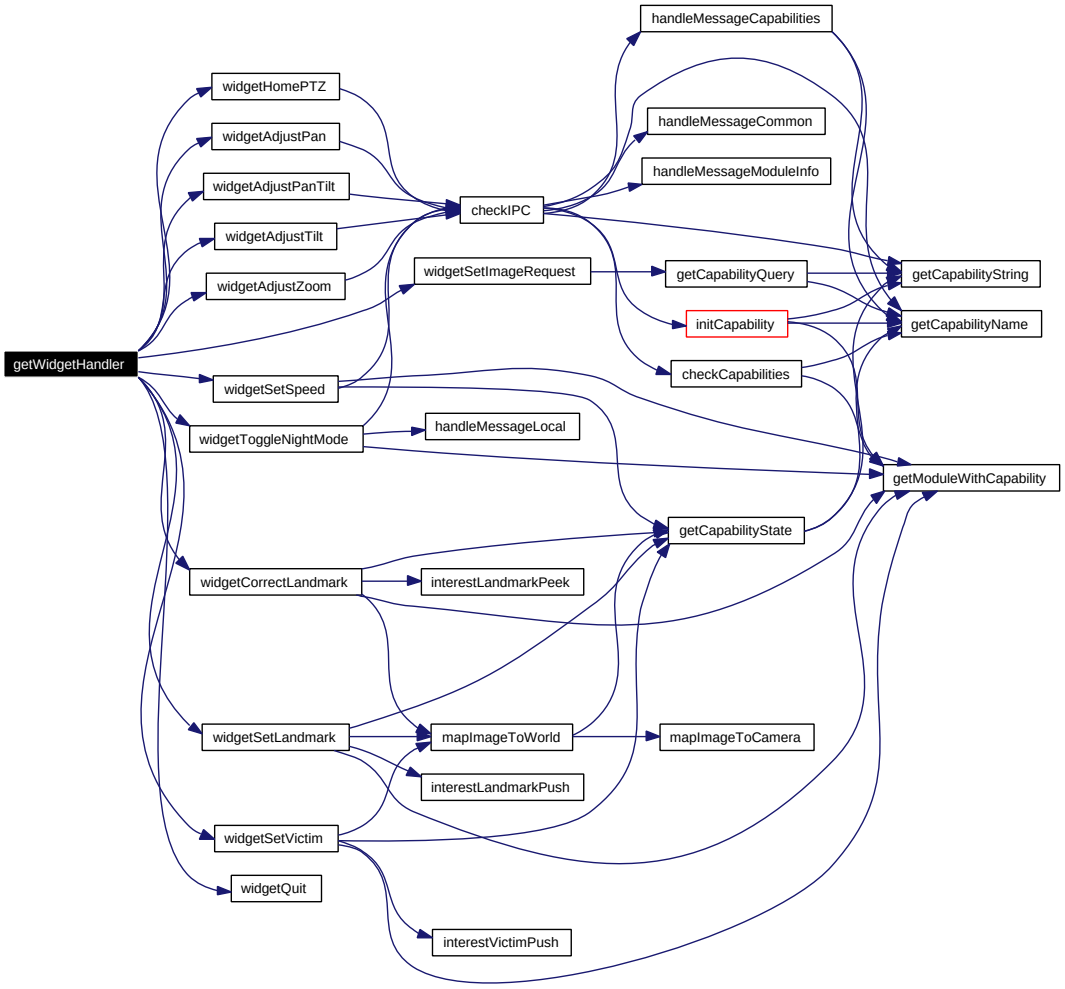
Author:
Nicolas Ward '05

Definition at line 20 of file widget.c.

References widgetAdjustPan(), widgetAdjustPanTilt(), widgetAdjustTilt(), widgetAdjustZoom(), widgetCorrectLandmark(), widgetHomePTZ(), widgetQuit(), widgetSetImageRequest(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

Referenced by parseWidget().

Here is the call graph for this function:



B.11.5.36 void handleEvent (SDL_Event * *event*, Rune * *rune*)

Processes SDL events and passes them off to the appropriate event handler.

Parameters:

← ***event*** The SDL event that triggered the handler.

← ***rune*** The **Rune**(p. 40) data structure.

Author:

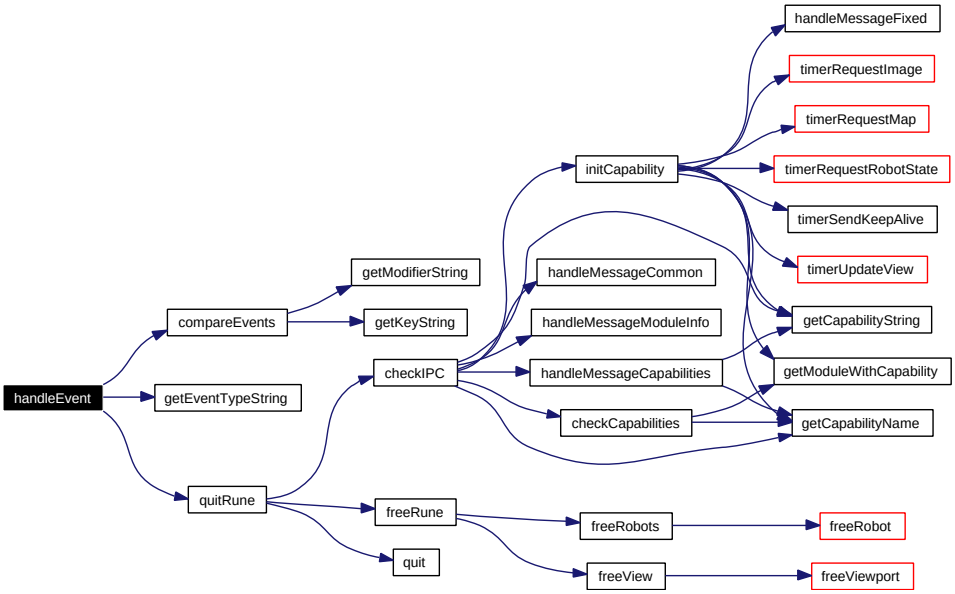
Nicolas Ward '05

Definition at line 192 of file event.c.

References compareEvents(), getEventTypeString(), and quitRune().

Referenced by main(), and runRune().

Here is the call graph for this function:



B.11.5.37 void handleMessageCapabilities (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles Robomon capability listing messages.

Based on capHandler in the rmon-control interface.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to a capabilities data structure.

Author:

Nicolas Ward ’05
Fritz Heckel ’05

Todo

Determine if not freeing causes a small memory leak.

Bug

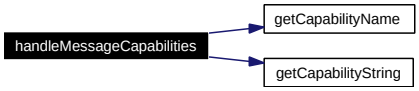
Freeing the capabilities first causes a segfault.

Definition at line 25 of file handler.c.

References getCapabilityName(), getCapabilityString(), Robot::haveCaps, and Robot::wantCaps.

Referenced by checkIPC().

Here is the call graph for this function:



B.11.5.38 void handleMessageCommon (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles GCM common command messages. If necessary, **Rune**(p. 40) will respond with the appropriate behavior.

Based on commonHandler in the skeleton module.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05
Fritz Heckel '05

Definition at line 100 of file handler.c.

References Rune::data, and Rune::info.

Referenced by checkIPC().

B.11.5.39 void handleMessageFixed (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Processes fixed-length IPC messages from a robot module and passes them off to the appropriate **Visualizer**(p. 48), based on the module's capabilities and the message type.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The incoming fixed-length IPC message received from one of the robot modules.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Definition at line 165 of file handler.c.

References Robot::context, Visualizer::data, Visualizer::function, Visualizer::message, Capability::robot, Viewport::robot, Robot::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by initCapability().

B.11.5.40 void handleMessageLocal (Widget * *widget*, char * *message*, void * *data*)

Passes arbitrary data off to the appropriate **Visualizer**(p. 48), based on the originating **Widget**(p. 51).

Parameters:

← ***widget*** The widget that sent this message

← ***message*** The string name of the message that was sent.

← ***data*** The arbitrary data associated with the message.

Author:

Nicolas Ward '05

Definition at line 238 of file handler.c.

References Visualizer::data, Visualizer::function, Visualizer::message, Viewport::robot, View::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by widgetToggleNightMode().

B.11.5.41 void handleMessageModuleInfo (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Handles Robomon module information messages.

Based on modHandler in the rmon-control interface.

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The GCM common command contained in the IPC message.

← ***clientData*** A pointer to a module information data structure.

Author:

Nicolas Ward '05
Fritz Heckel '05

Definition at line 309 of file handler.c.

Referenced by checkIPC().

B.11.5.42 void handleMessageVariable (MSG_INSTANCE *msgInstance*, void * *callData*, void * *clientData*)

Processes variable-length IPC messages from a robot module and passes them off to the appropriate message handler, which should be a **Visualizer**(p. 48).

Parameters:

← ***msgInstance*** A unique IPC message ID.

← ***callData*** The incoming variable-length IPC message received from one of the robot modules.

← ***clientData*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Definition at line 364 of file handler.c.

References Robot::context, Visualizer::data, Visualizer::function, Visualizer::message, Capability::robot, Viewport::robot, Robot::rune, Rune::running, Viewport::updated, Rune::view, Visualizer::viewport, View::viewports, Viewport::visible, and Viewport::visualizer.

Referenced by timerRequestImage(), timerRequestMap(), and timerRequestRobotState().

B.11.5.43 void initCapability (Capability * *capability*)

Initializes a **Capability**(p. 23) data structure based on its GCM_Capability enumerated type.

This function is called by **checkIPC()**(p. 103) after a successful connection to IPC central is made. It expects that all types have been defined properly.

This function handles the initialization of all capabilities that **Rune**(p. 40) knows about. For extensibility, it might be better to break the initialization step into multiple functions, and have this function call those functions as necessary. It might make the code more readable.

Parameters:

← ***capability*** A pointer to the **Capability**(p. 23) being initialized.

Todo

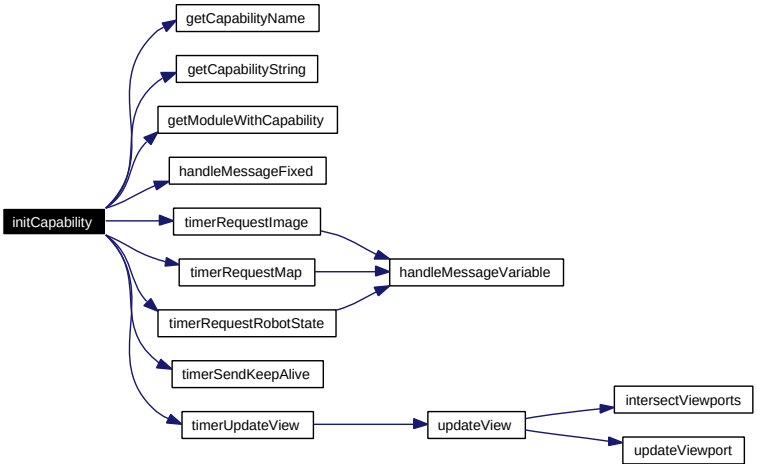
Maybe break this up into separate initializer functions?

Definition at line 531 of file capability.c.

References getCapabilityName(), getCapabilityString(), getModuleWithCapability(), CommonRequest::handled, ImageRequest::handled, handleMessageFixed(), Capability::nTimers, Capability::query, R_ALIVE_INTERVAL, R_IMAGE_INTERVAL, R_MAP_INTERVAL, R_NAV_INTERVAL, CommonRequest::request, ImageRequest::request, Capability::state, timerRequestImage(), timerRequestMap(), timerRequestRobotState(), Capability::timers, timerSendKeepAlive(), and timerUpdateView().

Referenced by checkIPC().

Here is the call graph for this function:



B.11.5.44 void initJoysticks (Rune * *rune*)

Detects and initializes physical joysticks attached to the client computer.

Parameters:

↔ ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

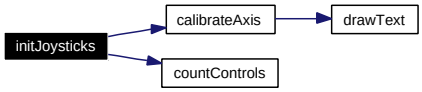
Nicolas Ward '05

Definition at line 189 of file joystick.c.

References calibrateAxis(), countControls(), Control::joystick, JOYSTICK_CONTROL_AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, and JOYSTICK_CONTROL_HAT_SWITCH.

Referenced by main(), and runRune().

Here is the call graph for this function:



B.11.5.45 InterestPoint* interestLandmarkPeek (InterestPoints * *points*)

Gets the current landmark.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

Returns:

A pointer to the topmost **InterestPoint**(p. 33).

Author:

Nicolas Ward '05

Definition at line 46 of file interest.c.

Referenced by widgetCorrectLandmark().

B.11.5.46 void interestLandmarkPush (InterestPoints * *points*, InterestPoint * *landmark*)

Adds a new landmark.

Landmarks are used as waypoints in robot navigation, and can be used to correct erroneous robot odometry. They are stored in a push-only stack.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

← ***landmark*** The **InterestPoint**(p. 33) data structure being added.

Author:

Nicolas Ward '05

Definition at line 64 of file interest.c.

Referenced by widgetSetLandmark().

B.11.5.47 InterestPoint* interestVictimPeek (InterestPoints * *points*)

Gets the current victim.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

Returns:

A pointer to the topmost **InterestPoint**(p. 33).

Author:

Nicolas Ward '05

Definition at line 100 of file interest.c.

B.11.5.48 void interestVictimPush (InterestPoints * *points*, InterestPoint * *victim*)

Adds a new victim.

Victims are used as waypoints in robot navigation, and can be used to correct erroneous robot odometry. They are stored in a push-only stack.

Parameters:

← ***points*** The **InterestPoints**(p. 34) data structure being examined.

← ***victim*** The **InterestPoint**(p. 33) data structure being added.

Author:

Nicolas Ward '05

Definition at line 118 of file interest.c.

Referenced by widgetSetVictim().

B.11.5.49 bool intersectViewports (Viewport * *viewport1*, Viewport * *viewport2*)

Checks if one **Viewport**(p. 45) overlaps another, to determine if the overlapped **Viewport**(p. 45) needs to be redrawn.

Checks if the second (overlapping) **Viewport**(p. 45) is in front, and if it is somewhere inside the first (overlapped) Viewport's bounding box.

Parameters:

← ***viewport1*** A pointer to the overlapped **Viewport**(p. 45).

← ***viewport2*** A pointer to the overlapping **Viewport**(p. 45).

Returns:

True if there is any overlap.

Author:

Nicolas Ward '05

Definition at line 91 of file viewport.c.

Referenced by updateView().

B.11.5.50 void mapCameraToImage (double *cameraSpaceX*, double *cameraSpaceY*, GCM_CameraState * *camera*, double *imageSpaceW*, double *imageSpaceH*, double * *imageSpaceX*, double * *imageSpaceY*)

Performs a coordinate mapping from an (x,y) position relative to the robot's axes to an (x,y) pixel value in an image.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

→ ***cameraSpaceX*** The x-coordinate in camera space in meters.

→ ***cameraSpaceY*** The y-coordinate in camera space in meters.

← ***camera*** The camera whose coordinate space is being used.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

→ ***imageSpaceX*** The x-coordinate in image space in pixels.

→ ***imageSpaceY*** The y-coordinate in image space in pixels.

Author:

Nicolas Ward ’05

```
*
* Convert from 2-D cartesian in camera space to 3-D polar in camera space
*  Angles /_hc and /_vc are measured in radians from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*  Ranges Rxy, and Ryz are measured in meters from the camera mountpoint
*    cameraSpaceRadiusH is in the ground plane
*    cameraSpaceRadiusV is in the vertical plane through the robot
*  Point (x,y) is measured in meters from below the camera's center point
*    cameraSpaceX is positive right
*    cameraSpaceY is positive forwards
*  Camera height is measured in meters from the ground plane
*    height is positive up
*
*    (top view)                        (side view)
*             . world point          camera ._______
*         |  /                              |\) -cameraSpaceAngleV
*       y|~/ Rxy                          h | \ Ryz
*         |/__    ~cameraSpaceAngleH        |  \
*         ' x                               |__(\
*  camera                                        'world point
*  center                      by Alternate Interior Angle theorem
*
*



*
* Convert from 3-D polar in camera space to radians in image space
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  Angles /_p and /_t are measured to the center of the camera's image
*    pan is positive left (CCW from straight ahead)
*    tilt is positive up (CCW from downwards vertical)
*  Angles /_hc and /_vc are measured from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*
```

```
*  camera .            \   |
*         |\            \  |
*         |~\ /_t    /_p |
*         |  \           \|
*         |   \           'camera
*      (side view)     (top view)
*
*


*
* Convert from pixels to radians in image space
*  Pixel (x,y) is measured from the upper left of the image
*    imageSpaceX is positive right
*    imageSpaceY is positive down
*  The pixel dimensions of the image are imageSpaceW by imageSpaceH
*    The image center is at (w/2,h/2)
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  The angular dimensions of the image are HFOV by VFOV
*
*     (side view)      (top view)            (front view)
*         /|  \           /|  \              _________________
*        / |  |          / |  |            |  x              | \
*  VFOV /  |  |    HFOV /  |  |           y|        | /_v    | |
*      <)  |  }h       <)  |  }w           |     ___|        | }h
*       \  |  |          \  |  |           |    /_h 'center  | |
*        \ |  |           \ |  |           |                 | |
*         \|  /            \|  /           |_________________| /
* image plane     image plane
*                                          \________w________/
*
*
```

Definition at line 283 of file widget.c.

Referenced by mapWorldToImage(), and visualizeGroundPlane().

B.11.5.51 void mapImageToCamera (double *imageSpaceX*, double *imageSpaceY*, double *imageSpaceW*, double *imageSpaceH*, GCM_CameraState * *camera*, double * *cameraSpaceX*, double * *cameraSpaceY*)

Performs a coordinate mapping from an (x,y) pixel value in an image to an (x,y) position value relative to the ground plane projection of the camera's axes.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

← ***imageSpaceX*** The x-coordinate in image space in pixels.

← ***imageSpaceY*** The y-coordinate in image space in pixels.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

← ***camera*** The camera whose coordinate space is being used.

→ ***cameraSpaceX*** The x-coordinate in camera space in meters.

→ ***cameraSpaceY*** The y-coordinate in camera space in meters.

Author:
Nicolas Ward ’05

```
*
* Convert from pixels to radians in image space
*  Pixel (x,y) is measured from the upper left of the image
*    imageSpaceX is positive right
*    imageSpaceY is positive down
*  The pixel dimensions of the image are imageSpaceW by imageSpaceH
*    The image center is at (w/2,h/2)
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  The angular dimensions of the image are HFOV by VFOV
*
*     (side view)      (top view)            (front view)
*         /|  \           /|  \              _________________
*        / |  |          / |  |             |  x              | \
*  VFOV /  |  |    HFOV /  |  |           y|        | /_v    | |
*      <)  |  }h       <)  |  }w           |     ___|        | }h
*       \  |  |         \  |  |            |    /_h ‘center  | |
*        \ |  |          \ |  |            |                 | |
*         \|  /           \|  /            |_________________| /
* image plane     image plane
*                                          \________w________/
*
*
```

```
*
* Convert from radians in image space to 3-D polar in camera space
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  Angles /_p and /_t are measured to the center of the camera’s image
*    pan is positive left (CCW from straight ahead)
*    tilt is positive up (CCW from downwards vertical)
*  Angles /_hc and /_vc are measured from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*
*  camera .            \   |
*         |\            \  |
*         |~\ /_t    /_p |
*         |  \           \|
```

```
*         |   \            ‘camera
*      (side view)     (top view)
*
*


*
* Convert from 3-D polar in camera space to 2-D cartesian in camera space
*  Angles /_hc and /_vc are measured in radians from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*  Ranges Rxy, and Ryz are measured in meters from the camera mountpoint
*    cameraSpaceRadiusH is in the ground plane
*    cameraSpaceRadiusV is in the vertical plane through the robot
*  Point (x,y) is measured in meters from below the camera’s center point
*    cameraSpaceX is positive right
*    cameraSpaceY is positive forwards
*  Camera height is measured in meters from the ground plane
*    height is positive up
*
*    (top view)                        (side view)
*             . world point          camera ._______
*         |  /                              |\) -cameraSpaceAngleV
*        y|~/ Rxy                         h | \ Ryz
*         |/__    ~cameraSpaceAngleH       |  \
*         ’ x                               |__(\
*  camera                                        ‘world point
*  center                       by Alternate Interior Angle theorem
*
*
```

Definition at line 82 of file widget.c.

Referenced by mapImageToWorld().

B.11.5.52 void mapImageToWorld (double *imageSpaceX*, double *imageSpaceY*, double *imageSpaceW*, double *imageSpaceH*, Robot * *robot*, double * *worldSpaceX*, double * *worldSpaceY*)

Performs a coordinate mapping from an (x,y) pixel value in an image to an (x,y) position value on the global ground plane.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

← ***imageSpaceX*** The x-coordinate in image space in pixels.

← ***imageSpaceY*** The y-coordinate in image space in pixels.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

← ***robot*** The robot whose coordinate space is being used.

→ ***worldSpaceX*** The x-coordinate in world space in meters.

→ ***worldSpaceY*** The y-coordinate in world space in meters.

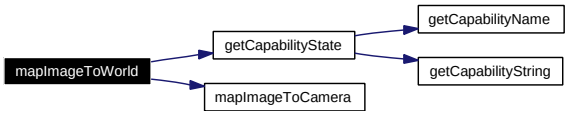
Author:
Nicolas Ward '05

Definition at line 208 of file widget.c.

References getCapabilityState(), and mapImageToCamera().

Referenced by widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

Here is the call graph for this function:



B.11.5.53 void mapWorldToImage (double *worldSpaceX*, double *worldSpaceY*, Robot * *robot*, double *imageSpaceW*, double *imageSpaceH*, double * *imageSpaceX*, double * *imageSpaceY*)

Performs a coordinate mapping from an (x,y) position value on the global ground plane to an (x,y) pixel value in an image.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

← ***worldSpaceX*** The x-coordinate in world space in meters.

← ***worldSpaceY*** The y-coordinate in world space in meters.

← ***robot*** The robot whose coordinate space is being used.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

→ ***imageSpaceX*** The x-coordinate in image space in pixels.

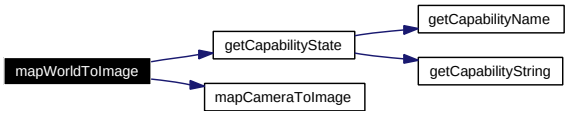
→ ***imageSpaceY*** The y-coordinate in image space in pixels.

Author:
Nicolas Ward '05

Definition at line 407 of file widget.c.

References getCapabilityState(), and mapCameraToImage().

Here is the call graph for this function:



B.11.5.54 void parseControl (Control ** *controlPtr*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <control> element from an XML configuration file.

Parameters:

→ ***controlPtr*** A pointer to the **Control**(p. 26) data structure being allocated by this function.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

Nicolas Ward '05

Definition at line 22 of file xml.c.

References Control::calibrate, getJoystickControlType(), Control::index, Control::invert, JOYSTICK_CONTROL_AXIS, Control::max, Control::min, Control::name, and Control::type.

Referenced by parseJoystick().

Here is the call graph for this function:

B.11.5.55 void parseDocument (char * *filename*, Rune * *rune*)

Parses an XML document, validates it, and loads the **Rune**(p. 40) data structure.

Parameters:

← ***filename*** The filename of the XML configuration file.

↔ ***rune*** The **Rune**(p. 40) data structure.

Author:

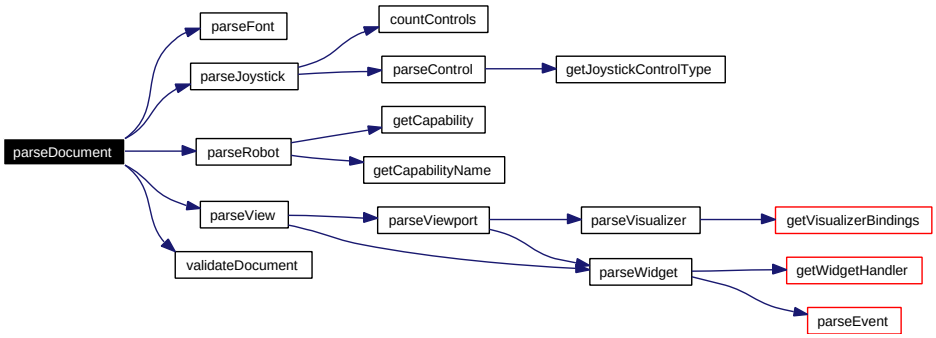
Nicolas Ward '05

Definition at line 168 of file xml.c.

References parseFont(), parseJoystick(), parseRobot(), parseView(), and validateDocument().

Referenced by main(), and runRune().

Here is the call graph for this function:



B.11.5.56 void parseEvent (Event ∗∗ *eventPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses an <event> element from an XML configuration file.

Parameters:

→ ***eventPtr*** A pointer to the **Event**(p. 28) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

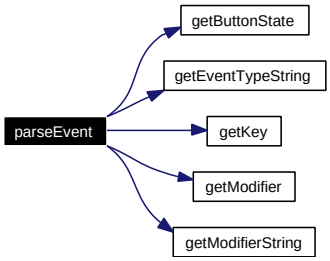
Nicolas Ward '05

Definition at line 319 of file xml.c.

References Joystick::config, Event::control, Joystick::controls, Event::event, getButtonState(), getEventTypeString(), getKey(), getModifier(), getModifierString(), Control::index, Event::joystick, JOYSTICK_CONTROL_AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, JOYSTICK_CONTROL_HAT_SWITCH, Control::name, Joystick::nControls, Event::options, and Control::type.

Referenced by parseWidget().

Here is the call graph for this function:



B.11.5.57 void parseFont (Font * *font*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <font> element from an XML state file.

Parameters:

→ ***font*** A pointer to the **Font**(p. 30) data structure being allocated by this function.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

Nicolas Ward '05

Definition at line 694 of file xml.c.

Referenced by parseDocument().

B.11.5.58 void parseJoystick (Joystick ** *joystickPtr*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <joystick> element from an XML configuration file.

Parameters:

→ ***joystickPtr*** A pointer to the **Joystick**(p. 35) data structure being allocated by this function.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

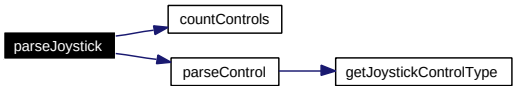
Nicolas Ward '05

Definition at line 742 of file xml.c.

References Joystick::config, Joystick::controls, countControls(), Joystick::joystick, Control::joystick, Joystick::nControls, and parseControl().

Referenced by parseDocument().

Here is the call graph for this function:



B.11.5.59 void parseRobot (Robot ** *robotPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <robot> element from an XML configuration file.

Parameters:

→ ***robotPtr*** A pointer to the **Robot**(p. 37) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

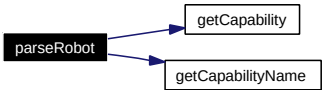
Nicolas Ward '05

Definition at line 820 of file xml.c.

References Capability::cap, getCapability(), getCapabilityName(), Robot::haveCaps, Robot::hostname, Robot::moduleInfo, Robot::name, Capability::query, Capability::ready, Capability::robot, Capability::state, and Robot::wantCaps.

Referenced by parseDocument().

Here is the call graph for this function:



B.11.5.60 void parseView (View ** *viewPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses the <view> element from an XML configuration file.

Parameters:

→ ***viewPtr*** A pointer to the **View**(p. 42) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

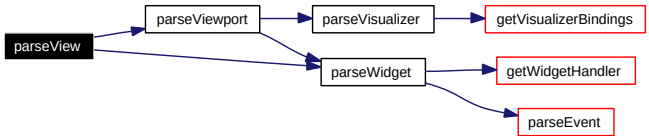
Author:
Nicolas Ward '05

Definition at line 935 of file xml.c.

References View::fullscreen, View::nViewports, View::nWidgets, parseViewport(), parseWidget(), Viewport::view, Widget::view, View::viewports, View::widgets, View::xsize, and View::ysize.

Referenced by parseDocument().

Here is the call graph for this function:



B.11.5.61 void parseViewport (Viewport ** *viewportPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <viewport> element from an XML configuration file.

Parameters:
- → ***viewportPtr*** A pointer to the **Viewport**(p. 45) data structure being allocated by this function.
- ↔ ***rune*** The **Rune**(p. 40) data structure.
- ← ***doc*** A pointer to the XML document being read.
- ← ***node*** A pointer to the current XML node.

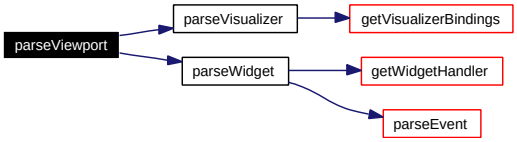
Author:
Nicolas Ward '05

Definition at line 1084 of file xml.c.

References Viewport::nWidgets, parseVisualizer(), parseWidget(), Viewport::robot, Viewport::transparency, Viewport::view, Widget::viewport, Visualizer::viewport, Viewport::visible, Viewport::visualizer, Viewport::widgets, Viewport::xpos, Viewport::xsize, View::xsize, Viewport::ypos, Viewport::ysize, View::ysize, and Viewport::zpos.

Referenced by parseView().

Here is the call graph for this function:



B.11.5.62 void parseVisualizer (Visualizer ** *visualizerPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <visualizer> element from an XML configuration file.

Parameters:

→ ***visualizerPtr*** A pointer to the **Visualizer**(p. 48) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

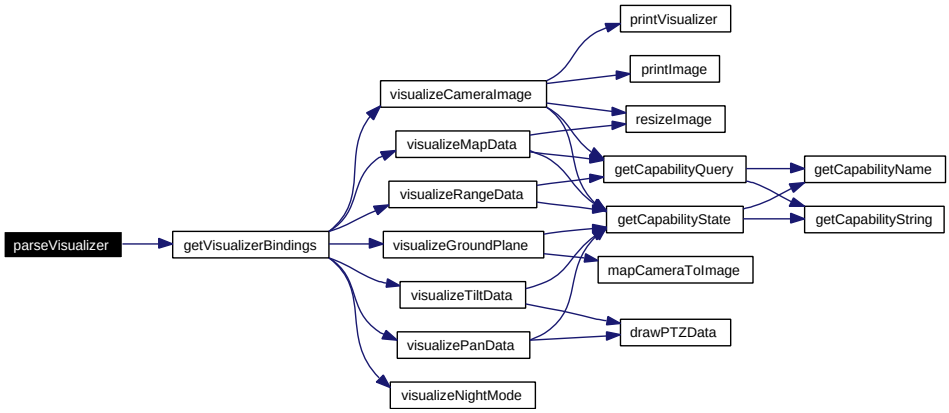
Nicolas Ward '05

Definition at line 1336 of file xml.c.

References Visualizer::data, Visualizer::function, getVisualizerBindings(), Visualizer::option, Visualizer::surface, Visualizer::viewport, Visualizer::xsize, and Visualizer::ysize.

Referenced by parseViewport().

Here is the call graph for this function:



B.11.5.63 void parseWidget (Widget ** *widgetPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <widget> element from an XML configuration file.

Parameters:

→ ***widgetPtr*** A pointer to the **Widget**(p. 51) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

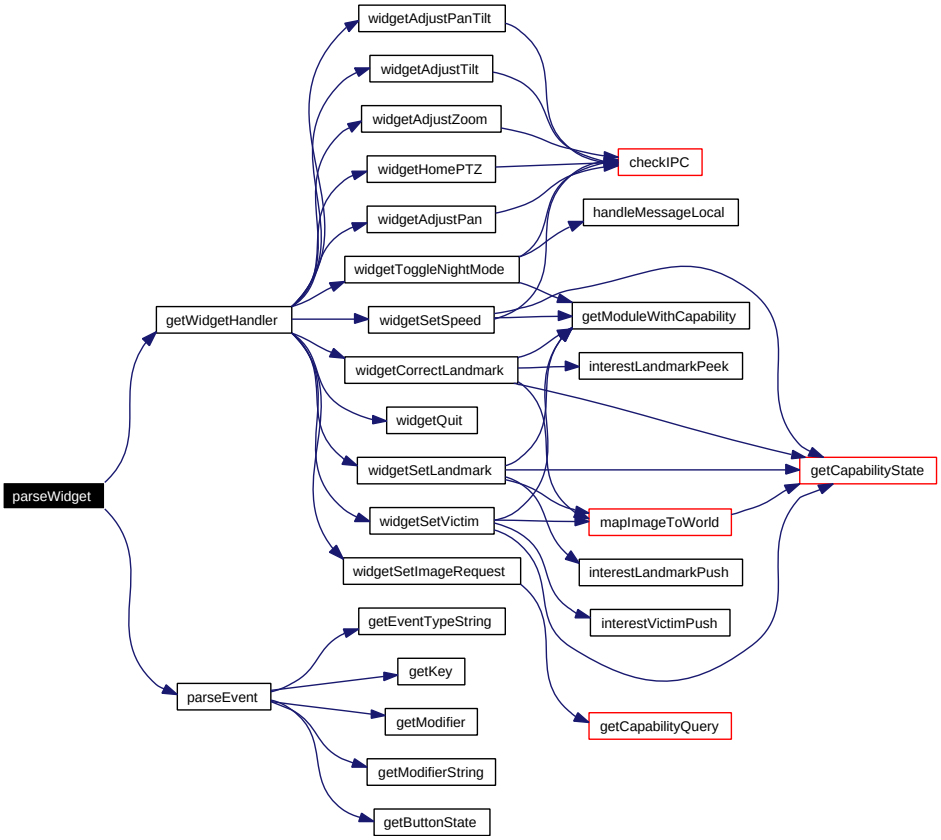
Author:
Nicolas Ward '05

Definition at line 1451 of file xml.c.

References Widget::events, getWidgetHandler(), Widget::handler, Widget::history, Widget::nEvents, parseEvent(), Widget::view, and Widget::viewport.

Referenced by parseView(), and parseViewport().

Here is the call graph for this function:



B.11.5.64 void printImage (GCM_Common_Image * *image*)

Outputs the text metadata that is stored in a GCM image data structure.

Parameters:

← ***image*** The image whose metadata will be output.

Author:
Nicolas Ward '05

Definition at line 207 of file visualizer.c.

Referenced by visualizeCameraImage().

B.11.5.65 void printVisualizer (Visualizer * *visualizer*)

Outputs the text metadata that is stored in a **Visualizer**(p. 48) data structure.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) whose metadata will be output.

Author:
Nicolas Ward '05

Definition at line 269 of file visualizer.c.

Referenced by visualizeCameraImage().

B.11.5.66 void quitRune (Rune * *rune*)

Closes IPC connections, deallocates data structures, and quits **Rune**(p. 40).

Called by the SDL main event loop when a quit event is received.

Parameters:
← ***rune*** A pointer to the **Rune**(p. 40) data strucutre.

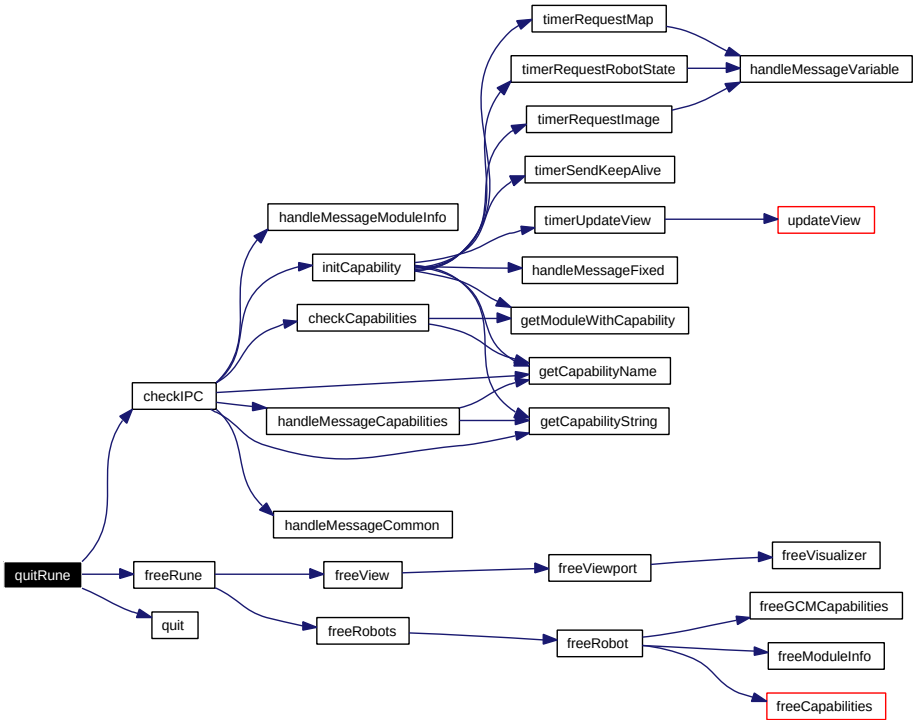
Author:
Nicolas Ward '05

Definition at line 261 of file rune.c.

References checkIPC(), Rune::data, freeRune(), quit(), and Rune::running.

Referenced by handleEvent().

Here is the call graph for this function:



B.11.5.67 void resizeImage (unsigned char * *input*, int *inputW*, int *inputH*, int *inputD*, unsigned char * *output*, int *outputW*, int *outputH*, int *outputD*)

Resizes an PPM-formatted image by interpolating the unsigned character data of the image.

This function cannot scale an image down because it does not do any sampling. It probably isn't particularly good or efficient at what it does either. Use sparingly.

This function does not handle color depths correctly, but it does work well enough to convert a PGM or PPM to a PPM.

Parameters:

← ***input*** The character data of the input image.

← ***inputW*** The width of the input image in pixels.

← ***inputH*** The height of the input image in pixels.

← ***inputD*** The color depth of the input image in bytes.

→ ***output*** A pre-allocated array where the character data of the output image will be written.

← ***outputW*** The width of the output image in pixels.

← ***outputH*** The height of the output image in pixels.

← ***outputD*** The color depth of the output image in bytes.

Author:

Nicolas Ward '05

Todo

Fix this function or find a replacement that's fast.

Definition at line 317 of file visualizer.c.

Referenced by visualizeCameraImage(), and visualizeMapData().

B.11.5.68 int runRune (int *argc*, char ** *argv*)

The main runtime function.

Checks command line arguments, allocates the **Rune**(p. 40) state data structures, parses the XML configuration file, calls SDL initialization functions, and executes Rune's main loop.

Parameters:

← ***argc*** The number of command line arguments.

← ***argv*** The array of command line argument strings.

Returns:

0 on successful execution, -1 on error.

Author:

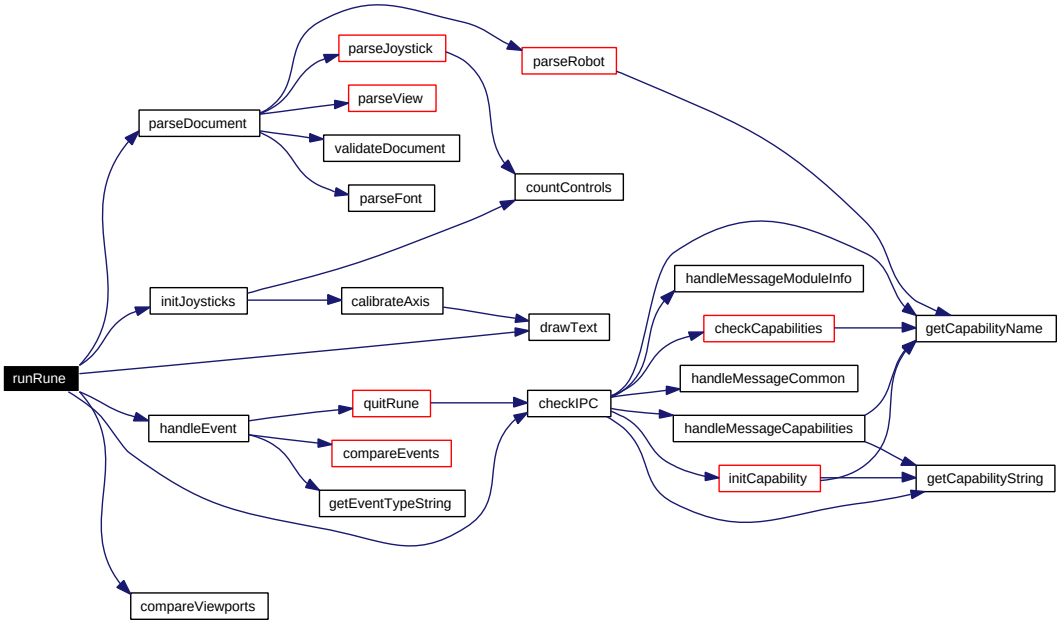
Nicolas Ward '05

Definition at line 53 of file rune.c.

References checkIPC(), compareViewports(), Rune::data, Rune::drawingFont, drawText(), Font::filename, Font::font, View::fullscreen, handleEvent(), Robot::hostname, Rune::info, initJoysticks(), Rune::nRobots, View::nViewports, parseDocument(), R_NAME, R_SDL_INIT_FLAGS, R_SDL_SURFACE_FLAGS, Rune::robots, Rune::running, Rune::screen, Font::size, Rune::view, View::viewports, View::xsize, and View::ysize.

Referenced by main().

Here is the call graph for this function:



B.11.5.69 Uint32 timerRequestImage (Uint32 *interval*, void * *param*)

Queries SVM for an image; called by an SDL timer associated with the VIS_VID capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

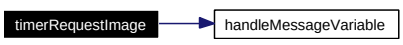
The unmodified timeout between calls of this function.

Definition at line 24 of file timer.c.

References Robot::context, ImageRequest::handled, handleMessageVariable(), Capability::query, ImageRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:



B.11.5.70 Uint32 timerRequestMap (Uint32 *interval*, void * *param*)

Queries SMM for a map image; called by an SDL timer associated with the PRO_MAP capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Definition at line 68 of file timer.c.

References Robot::context, CommonRequest::handled, handleMessageVariable(), Capability::query, CommonRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:

B.11.5.71 Uint32 timerRequestRobotState (Uint32 *interval*, void * *param*)

Queries Nav for the robot state; called by an SDL timer associated with the a NAV_* capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Definition at line 104 of file timer.c.

References Robot::context, CommonRequest::handled, handleMessageVariable(), Capability::query, CommonRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:

B.11.5.72 Uint32 timerSendKeepAlive (Uint32 *interval*, void * *param*)

Sends a keep-alive message; called by an SDL timer associated with the CON_REAL capability.

This message ensures that Robomon does not attempt to shutdown **Rune**(p. 40).

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Todo

Fix the module update

Definition at line 143 of file timer.c.

References Robot::context, Rune::data, Capability::robot, Rune::robots, Robot::rune, and Rune::running.

Referenced by initCapability().

B.11.5.73 Uint32 timerUpdateView (Uint32 *interval*, void * *param*)

Updates the main view; called by an SDL timer associated with the CON_REAL capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Definition at line 180 of file timer.c.

References Capability::robot, Robot::rune, Rune::running, and updateView().

Referenced by initCapability().

Here is the call graph for this function:

B.11.5.74 void updateView (Rune * *rune*)

Updates the entire **View**(p. 42), and all of its children, in back-to-front order.

The many #ifdefs in this function provide for compile-time hooks that select the drawing method being used. The constants are #defined in **rune.h**(p. 89).

Parameters:

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Figure out which surface method works best, and then strip out the unused code.

Todo

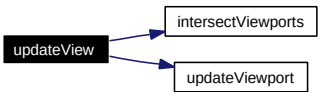
Optimize!

Definition at line 131 of file view.c.

References intersectViewports(), and updateViewport().

Referenced by timerUpdateView().

Here is the call graph for this function:



B.11.5.75 void updateViewport (Viewport * *viewport*, Rune * *rune*)

Updates a Viewport's display with newly visualized data.

The many #ifdefs in this function provide for compile-time hooks that select the drawing method being used. The constants are #defined in **rune.h**(p. 89).

Parameters:

← ***viewport*** The **Viewport**(p. 45) being updated.

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Figure out which surface method works best, and then strip out the unused code.

Todo
Optimize!

Definition at line 112 of file viewport.c.

References BMASK, Visualizer::surface, and Viewport::visualizer.

Referenced by updateView().

B.11.5.76 void validateDocument (char * *filename*, xmlDocPtr *doc*)

Validates an XML document using a parsed XML schema.

Parameters:
← ***filename*** The filename of the XML schema file.
← ***doc*** A pointer to the XML document being read.

Author:
Nicolas Ward '05

Definition at line 1542 of file xml.c.

Referenced by parseDocument().

B.11.5.77 void visualizeCameraImage (Visualizer * *visualizer*)

Uses the appropriate GCM RLE decoding function to convert the compressed image data sent over IPC into a raw PPM-formatted color image based on the specified size and quality.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) containing the image data.

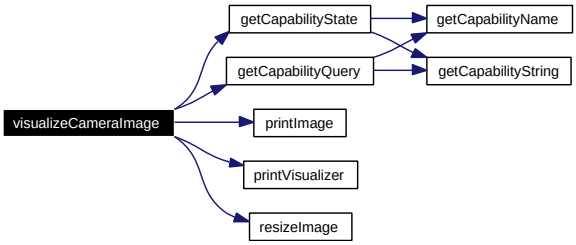
Author:
Nicolas Ward '05

Definition at line 376 of file visualizer.c.

References AMASK, BMASK, getCapabilityQuery(), getCapabilityState(), GMASK, ImageRequest::handled, printImage(), printVisualizer(), resizeImage(), and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.78 void visualizeGroundPlane (Visualizer * *visualizer*)

Uses incoming camera state data sent over IPC to calculate and draw indicators that approximate a ground plane in the camera's view.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

Author:

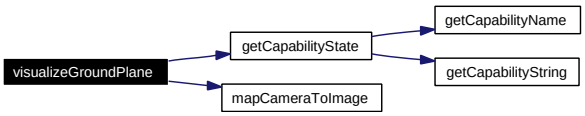
Nicolas Ward '05

Definition at line 566 of file visualizer.c.

References BMASK, getCapabilityState(), GMASK, mapCameraToImage(), R_SDL_-SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.79 void visualizeMapData (Visualizer * *visualizer*)

Uses the appropriate GCM RLE decoding function to convert the compressed map data sent over IPC into a raw PPM-formatted color image based on the mapping of different types of interest points.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the map data.

Author:

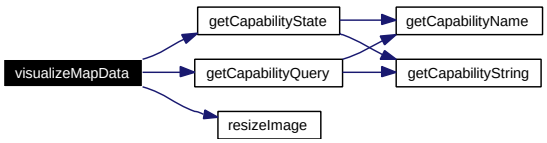
Nicolas Ward '05

Definition at line 672 of file visualizer.c.

References AMASK, BMASK, getCapabilityQuery(), getCapabilityState(), GMASK, Common-Request::handled, resizeImage(), and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.80 void visualizeNightMode (Visualizer * *visualizer*)

Responds to a local R_TOGGLE_NIGHT_MODE message to display an icon as needed.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) that will draw the icon.

Author:

Nicolas Ward '05

Definition at line 879 of file visualizer.c.

References BMASK, GMASK, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

B.11.5.81 void visualizePanData (Visualizer * *visualizer*)

Draws a bar indicating current pan position relative to the center line, as well as the current zoom setting in terms of angular field-of-view.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

Author:

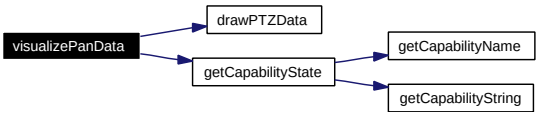
Nicolas Ward '05

Definition at line 962 of file visualizer.c.

References drawPTZData(), and getCapabilityState().

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.82 void visualizeRangeData (Visualizer * *visualizer*)

Draws a range data representation. Assumes a circular robot with radial sensors; in other words, a Magellan.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the robot state data.

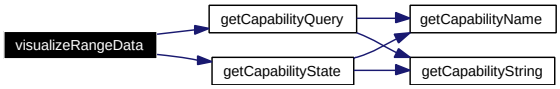
Author:
Nicolas Ward '05

Definition at line 1021 of file visualizer.c.

References BMASK, FILLED_PIE, getCapabilityQuery(), getCapabilityState(), GMASK, CommonRequest::handled, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.83 void visualizeTiltData (Visualizer * *visualizer*)

Draws a bar indicating current tilt position relative to the center line, as well as the current zoom setting in terms of angular field-of-view.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

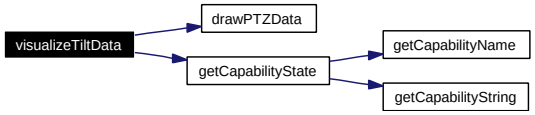
Author:
Nicolas Ward '05

Definition at line 1222 of file visualizer.c.

References drawPTZData(), and getCapabilityState().

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.11.5.84 void widgetAdjustPan (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the pan state of the robot's main camera based on input from a button or key.

Parameters:
← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

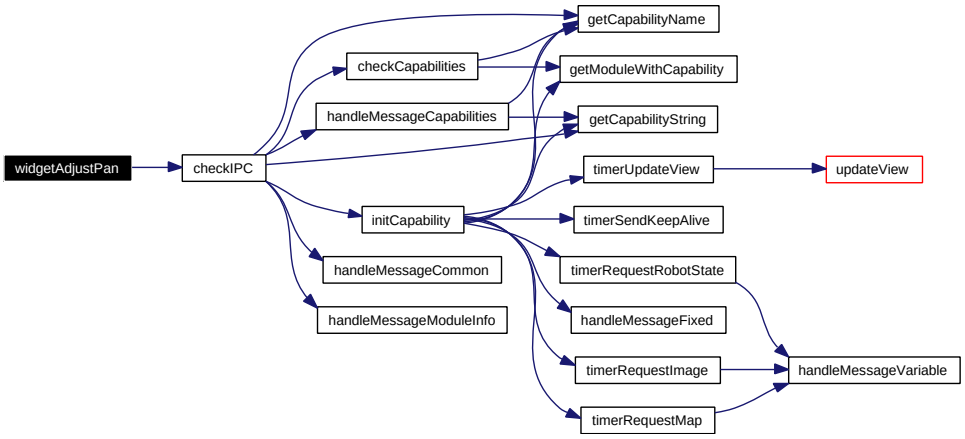
Author:
Nicolas Ward '05

Definition at line 481 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.85 void widgetAdjustPanTilt (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the pan-tilt state of the robot's main camera based on input from a hat switch.

Parameters:
← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

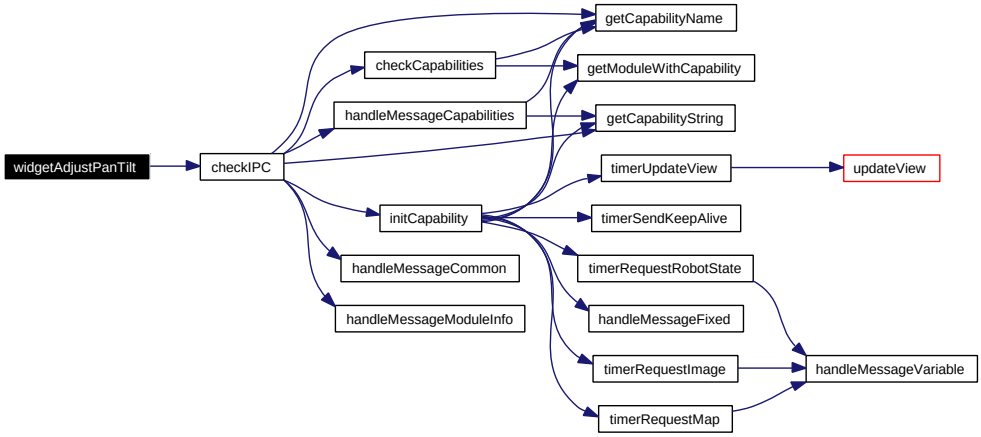
Author:
Nicolas Ward '05

Definition at line 531 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.86 void widgetAdjustTilt (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the tilt state of the robot's main camera based on input from a button or key.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

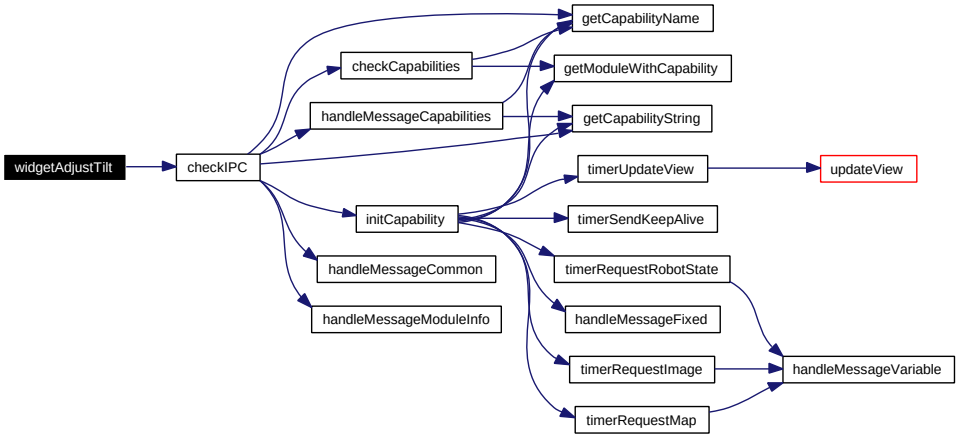
Nicolas Ward '05

Definition at line 585 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.87 void widgetAdjustZoom (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the zoom state of the robot's main camera based on input from a button or key.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

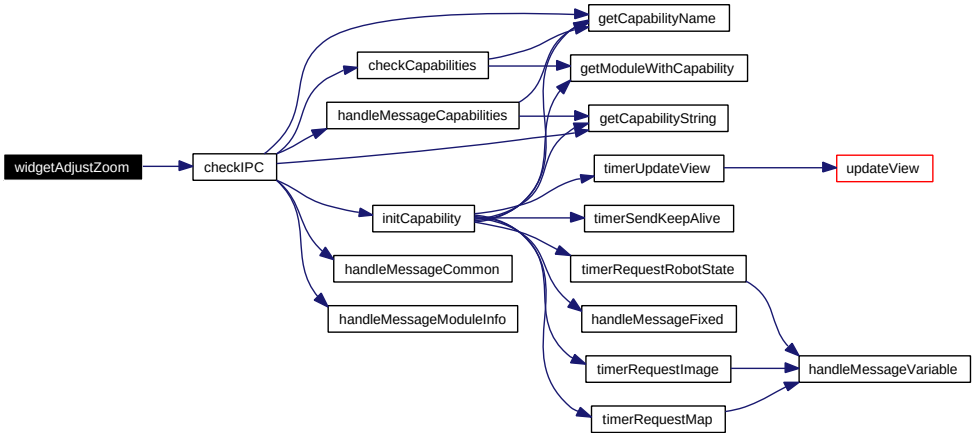
Nicolas Ward '05

Definition at line 635 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.88 void widgetCorrectLandmark (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Performs an odometry correction based on the current landmark.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

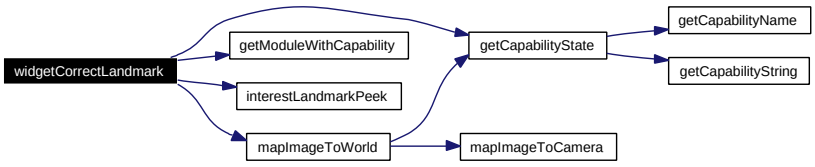
Nicolas Ward '05

Definition at line 684 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestLandmarkPeek(), mapImageToWorld(), InterestPoints::nLandmarks, Viewport::robot, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.89 void widgetHomePTZ (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Homes the PTZ state of the robot's main camera based on any input.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

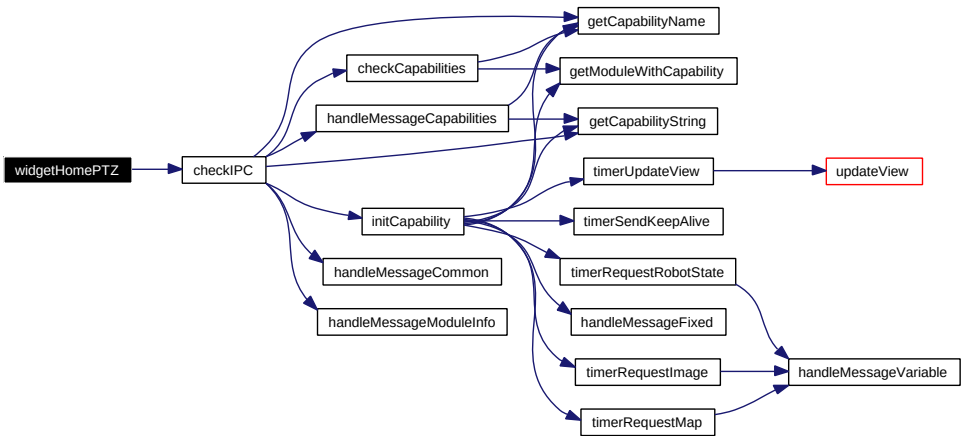
Nicolas Ward '05

Definition at line 806 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.90 void widgetQuit (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Pushes an SDL_QUIT event onto the queue.

The actual SDL_QUIT event is handled in handleEvent, which calls the quit function, cleans up, and actually quits **Rune**(p. 40).

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

Nicolas Ward '05

Definition at line 838 of file widget.c.

References InterestPoint::type.

Referenced by getWidgetHandler().

B.11.5.91 void widgetSetImageRequest (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Changes the type, size, and quality of the image being requested.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

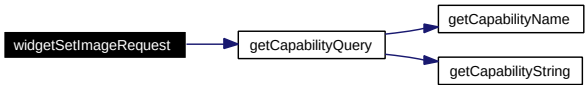
Author:
Nicolas Ward '05

Definition at line 857 of file widget.c.

References getCapabilityQuery(), Control::invert, Control::max, Control::min, test, and Control::type.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.92 void widgetSetLandmark (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Stores a landmark for future use.

Parameters:
← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

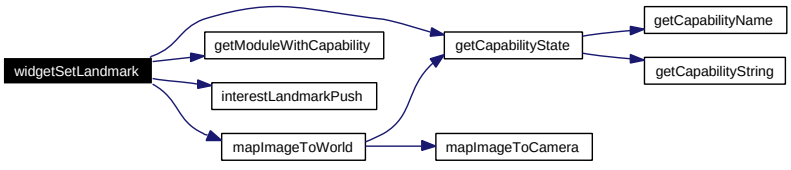
Author:
Nicolas Ward '05

Definition at line 935 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestLandmarkPush(), mapImageToWorld(), InterestPoints::nLandmarks, Viewport::robot, InterestPoint::type, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.93 void widgetSetSpeed (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Sets rotation and translation speeds for the robot, based on joystick axes.

Normally has the functionality to home the robot's PTZ camera, but that is disabled until we integrate some SVM messages into GCM.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

Nicolas Ward '05

Todo

Remove dual-axis dependency check.

Todo

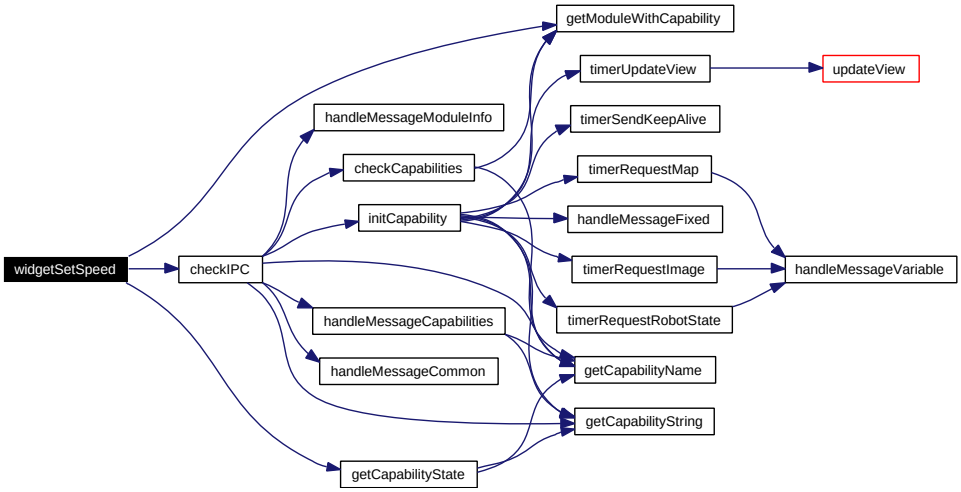
Remove or fix the automatic camera reset.

Definition at line 1162 of file widget.c.

References checkIPC(), Robot::context, getCapabilityState(), getModuleWithCapability(), Control::invert, Control::max, Control::min, Control::name, Viewport::robot, and test.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.94 void widgetSetVictim (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Stores a victim for future use.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

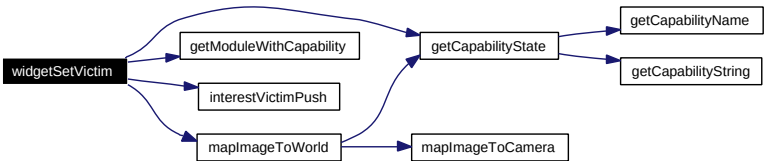
Nicolas Ward '05

Definition at line 1042 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestVictimPush(), mapImageToWorld(), InterestPoints::nVictims, Viewport::robot, InterestPoint::type, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.11.5.95 void widgetToggleNightMode (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Turns the night mode of a camera on or off.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

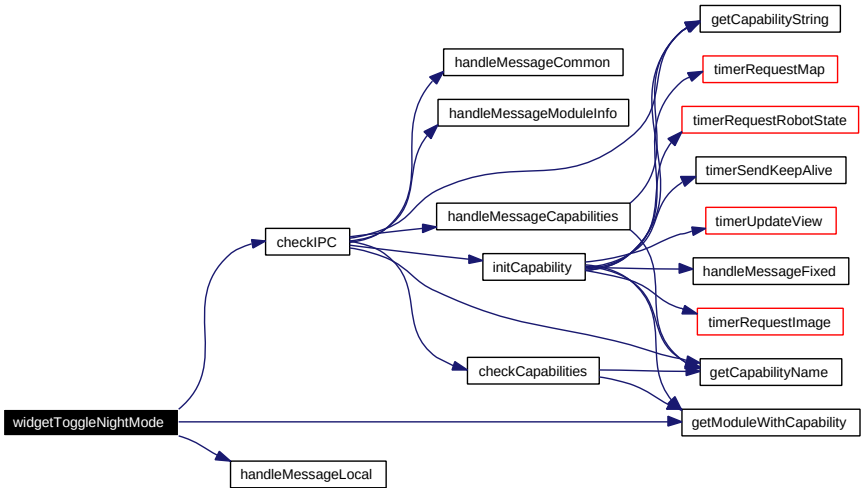
Nicolas Ward '05

Definition at line 1308 of file widget.c.

References checkIPC(), Robot::context, getModuleWithCapability(), handleMessageLocal(), R_-TOGGLE_NIGHT_MODE, and Viewport::robot.

Referenced by getWidgetHandler().

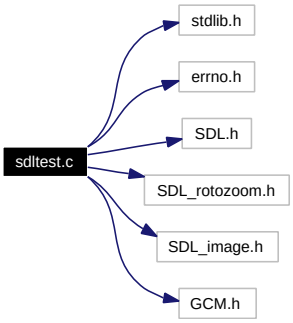
Here is the call graph for this function:



B.12 sdltest.c File Reference

```
#include <stdlib.h>
#include <errno.h>
#include <SDL.h>
#include <SDL_rotozoom.h>
#include <SDL_image.h>
#include <GCM.h>
```

Include dependency graph for sdltest.c:



Defines

- #define **ARG_SHIFT** 4
- #define **RMASK** 0xff000000
- #define **GMASK** 0x00ff0000
- #define **BMASK** 0x0000ff00
- #define **AMASK** 0x000000ff
- #define **SCREEN_WIDTH** 800
- #define **SCREEN_HEIGHT** 600
- #define **IMAGE_WIDTH** 160
- #define **IMAGE_HEIGHT** 120
- #define **SDL_SURFACE_FLAGS** SDL_HWSURFACE|SDL_RLEACCEL|SDL_HWPALETTE
- #define **USAGE** "usage: sdltest <**test** type> <max time> <interpolation> <ppm images> ...\n"
- #define **GRAYS** 256

Enumerations

- enum **TestType** { **TEST_COPY_ARRAY**, **TEST_MEMCPY_ARRAY**, **TEST_LOAD_ARRAY**, **TEST_SURFACE** }

Functions

- void **draw** (SDL_Surface **drawing, unsigned char *image)
- void **quit** (long start, int frames)
- int **main** (int argc, char **argv)

Variables

- **TestType test**
- SDL_Color **grays** [GRAYS]

B.12.1 Detailed Description

Performs various speed tests for SDL operations.

Author:
Nicolas Ward '05

Date:
2005.03.31

Definition in file **sdltest.c**.

B.12.2 Define Documentation

B.12.2.1 #define AMASK 0x000000ff

Definition at line 35 of file sdltest.c.

B.12.2.2 #define ARG_SHIFT 4

The number of non-image arguments, including the name of the program.

Todo

Switch to getopt.

Definition at line 28 of file sdltest.c.

Referenced by main().

B.12.2.3 #define BMASK 0x0000ff00

Definition at line 34 of file sdltest.c.

B.12.2.4 #define GMASK 0x00ff0000

Definition at line 33 of file sdltest.c.

B.12.2.5 #define GRAYS 256

The number of grayscale values allowed in a PGM-formatted image.

Definition at line 64 of file sdltest.c.

Referenced by draw(), and main().

B.12.2.6 #define IMAGE_HEIGHT 120

Definition at line 47 of file sdltest.c.

Referenced by draw(), and main().

B.12.2.7 #define IMAGE_WIDTH 160

Definition at line 46 of file sdltest.c.

Referenced by draw(), and main().

B.12.2.8 #define RMASK 0xff000000

Definition at line 32 of file sdltest.c.

B.12.2.9 #define SCREEN_HEIGHT 600

Definition at line 45 of file sdltest.c.

Referenced by main().

B.12.2.10 #define SCREEN_WIDTH 800

Definition at line 44 of file sdltest.c.

Referenced by main().

B.12.2.11 #define SDL_SURFACE_FLAGS SDL_HWSURFACE|SDL_RLEACCEL|SDL_HWPALETTE

Surface initialization flags for SDL.

Definition at line 54 of file sdltest.c.

Referenced by main().

B.12.2.12 #define USAGE "usage: sdltest <test type> <max time> <interpolation> <ppm images> ...\n"

Usage message printed on argument error.

Definition at line 59 of file sdltest.c.

Referenced by main().

B.12.3 Enumeration Type Documentation

B.12.3.1 enum TestType

An enumerated type which defines which SDL test is being performed.

Enumeration values:

TEST_COPY_ARRAY

TEST_MEMCPY_ARRAY

TEST_LOAD_ARRAY

TEST_SURFACE

Definition at line 71 of file sdltest.c.

B.12.4 Function Documentation

B.12.4.1 void draw (SDL_Surface ** *drawing*, unsigned char * *image*)

Converts an unsigned char array to an SDL_Surface.

Parameters:

→ ***drawing*** The surface resulting from the conversion.

← ***image*** The input grayscale image.

Author:

Nicolas Ward '05

Definition at line 342 of file sdltest.c.

References AMASK, BMASK, GMASK, GRAYS, grays, IMAGE_HEIGHT, IMAGE_WIDTH, RMASK, test, TEST_COPY_ARRAY, TEST_LOAD_ARRAY, and TEST_MEMCPY_ARRAY.

Referenced by main().

B.12.4.2 int main (int *argc*, char ** *argv*)

The main function; runs the entire specified test for the specified amount of time.

Parameters:

← ***argc*** The number of command line arguments.

← ***argv*** The array of command line argument strings.

Returns:

-1 on error, 0 otherwise.

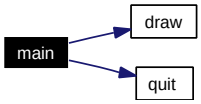
Author:

Nicolas Ward '05

Definition at line 103 of file sdltest.c.

References AMASK, ARG_SHIFT, BMASK, draw(), GMASK, grays, GRAYS, IMAGE_HEIGHT, IMAGE_WIDTH, quit(), RMASK, SCREEN_HEIGHT, SCREEN_WIDTH, SDL_SURFACE_FLAGS, test, TEST_COPY_ARRAY, TEST_LOAD_ARRAY, TEST_MEMCPY_ARRAY, TEST_SURFACE, and USAGE.

Here is the call graph for this function:



B.12.4.3 void quit (long *start*, int *frames*)

Calculates the average frame rate over the course of the run and exits.

Parameters:

← ***start*** The time in ticks at program start.

← ***frames*** The number of frames drawn since program start.

Author:

Nicolas Ward '05

Definition at line 392 of file sdltest.c.

Referenced by main(), and quitRune().

B.12.5 Variable Documentation

B.12.5.1 SDL_Color grays[GRAYS]

Color lookup table for grayscale values.

Definition at line 88 of file sdltest.c.

Referenced by draw(), and main().

B.12.5.2 TestType test

Stores the current test being performed.

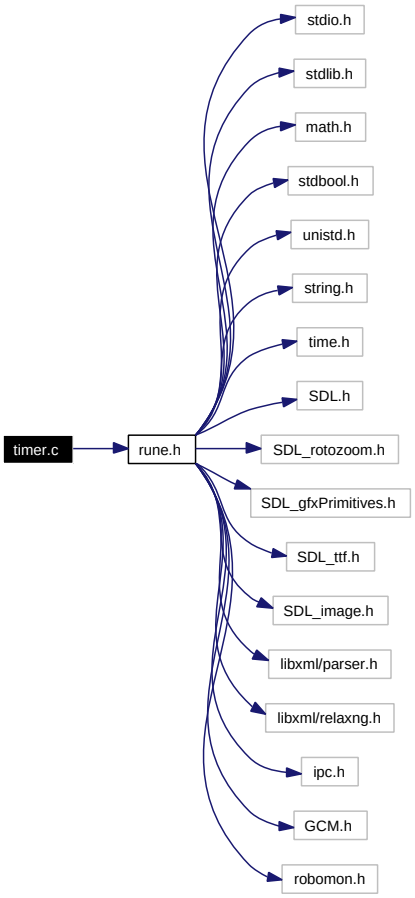
Definition at line 83 of file sdltest.c.

Referenced by draw(), main(), widgetSetImageRequest(), and widgetSetSpeed().

B.13 timer.c File Reference

`#include <rune.h>`

Include dependency graph for timer.c:



Functions

- Uint32 **timerRequestImage** (Uint32 interval, void ∗param)
- Uint32 **timerRequestMap** (Uint32 interval, void ∗param)
- Uint32 **timerRequestRobotState** (Uint32 interval, void ∗param)
- Uint32 **timerSendKeepAlive** (Uint32 interval, void ∗param)
- Uint32 **timerUpdateView** (Uint32 interval, void ∗param)

B.13.1 Detailed Description

Contains timer functions that are called regularly by **Rune**(p. 40) to perform some action associated with a **Capability**(p. 23).

These timers are called by the SDL main loop as scheduled when the timers are initialized to call these functions.

Author:

Nicolas Ward ’05

Date:
2005.03.31

Definition in file **timer.c**.

B.13.2 Function Documentation

B.13.2.1 Uint32 timerRequestImage (Uint32 *interval*, void * *param*)

Queries SVM for an image; called by an SDL timer associated with the VIS_VID capability.

Parameters:
← ***interval*** The timeout between calls of this function.
← ***param*** A pointer to the **Capability**(p. 23) data structure.

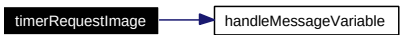
Returns:
The unmodified timeout between calls of this function.

Definition at line 24 of file timer.c.

References Robot::context, ImageRequest::handled, handleMessageVariable(), Capability::query, ImageRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:



B.13.2.2 Uint32 timerRequestMap (Uint32 *interval*, void * *param*)

Queries SMM for a map image; called by an SDL timer associated with the PRO_MAP capability.

Parameters:
← ***interval*** The timeout between calls of this function.
← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:
The unmodified timeout between calls of this function.

Definition at line 68 of file timer.c.

References Robot::context, CommonRequest::handled, handleMessageVariable(), Capability::query, CommonRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:

B.13.2.3 Uint32 timerRequestRobotState (Uint32 *interval*, void * *param*)

Queries Nav for the robot state; called by an SDL timer associated with the a NAV_* capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Definition at line 104 of file timer.c.

References Robot::context, CommonRequest::handled, handleMessageVariable(), Capability::query, CommonRequest::request, Capability::robot, Robot::rune, and Rune::running.

Referenced by initCapability().

Here is the call graph for this function:

B.13.2.4 Uint32 timerSendKeepAlive (Uint32 *interval*, void * *param*)

Sends a keep-alive message; called by an SDL timer associated with the CON_REAL capability.

This message ensures that Robomon does not attempt to shutdown **Rune**(p. 40).

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Todo

Fix the module update

Definition at line 143 of file timer.c.

References Robot::context, Rune::data, Capability::robot, Rune::robots, Robot::rune, and Rune::running.

Referenced by initCapability().

B.13.2.5 Uint32 timerUpdateView (Uint32 *interval*, void * *param*)

Updates the main view; called by an SDL timer associated with the CON_REAL capability.

Parameters:

← ***interval*** The timeout between calls of this function.

← ***param*** A pointer to the **Capability**(p. 23) data structure.

Returns:

The unmodified timeout between calls of this function.

Definition at line 180 of file timer.c.

References Capability::robot, Robot::rune, Rune::running, and updateView().

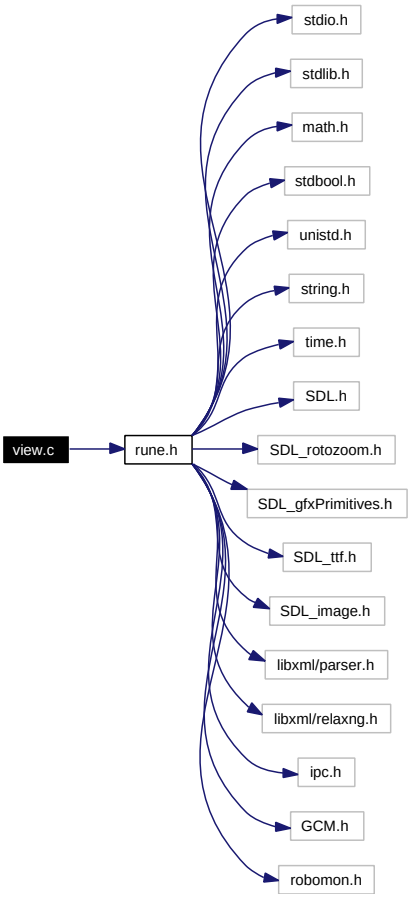
Referenced by initCapability().

Here is the call graph for this function:

B.14 view.c File Reference

`#include <rune.h>`

Include dependency graph for view.c:



Functions

- void **drawText** (SDL_Surface ∗target, TTF_Font ∗font, char ∗text)
- void **freeView** (**View** ∗view)
- void **updateView** (**Rune** ∗rune)

B.14.1 Detailed Description

Contains functions for configuring and displaying a **View**(p. 42) data structure.

Author:
Nicolas Ward ’05

Date:
2005.03.20

Definition in file **view.c**.

B.14.2 Function Documentation

B.14.2.1 void drawText (SDL_Surface * *target*, TTF_Font * *font*, char * *text*)

Draws some arbitrary text into a surface.

The text can be drawn in any TTF font. Currently the text is drawn in black on a grey background, which is then blitted into the middle of the target surface.

Parameters:

← ***target*** The surface onto which the text will be drawn.

← ***font*** The font used to render the text.

← ***text*** The string of text to be drawn.

Author:

Nicolas Ward '05

Todo

Add support for arbitrary text coloring.

Definition at line 25 of file view.c.

Referenced by calibrateAxis(), and runRune().

B.14.2.2 void freeView (View * *view*)

Frees a **View**(p. 42) data structure and all of its children.

Parameters:

← ***view*** The **View**(p. 42) structure being freed.

Author:

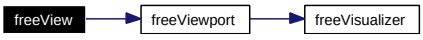
Nicolas Ward '05

Definition at line 94 of file view.c.

References freeViewport().

Referenced by freeRune().

Here is the call graph for this function:



B.14.2.3 void updateView (Rune * *rune*)

Updates the entire **View**(p. 42), and all of its children, in back-to-front order.

The many #ifdefs in this function provide for compile-time hooks that select the drawing method being used. The constants are #defined in **rune.h**(p. 89).

Parameters:

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Figure out which surface method works best, and then strip out the unused code.

Todo

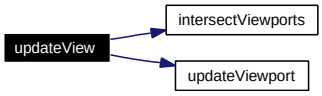
Optimize!

Definition at line 131 of file view.c.

References intersectViewports(), and updateViewport().

Referenced by timerUpdateView().

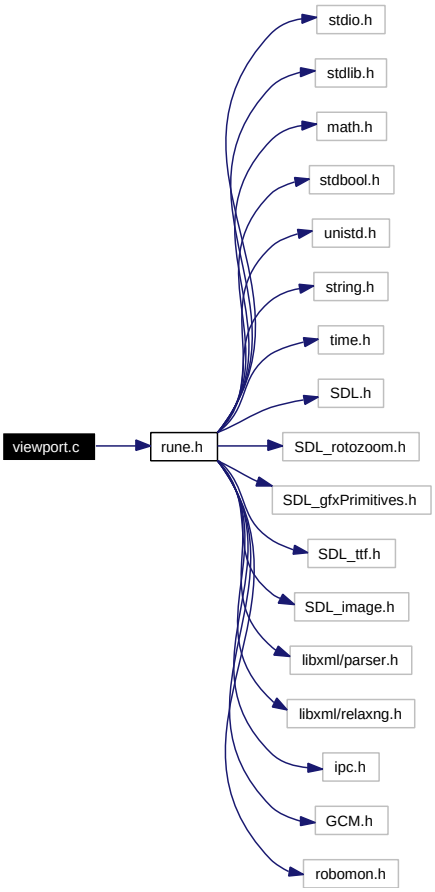
Here is the call graph for this function:



B.15 viewport.c File Reference

`#include <rune.h>`

Include dependency graph for viewport.c:



Functions

- int **compareViewports** (const void ∗viewportPtr1, const void ∗viewportPtr2)
- void **freeViewport** (**Viewport** ∗viewport)
- bool **intersectViewports** (**Viewport** ∗viewport1, **Viewport** ∗viewport2)
- void **updateViewport** (**Viewport** ∗viewport, **Rune** ∗rune)

B.15.1 Detailed Description

Contains functions for the configuring and displaying a **Viewport**(p. 45) data structure.

Author:
Nicolas Ward ’05

Date:
2005.03.20

Definition in file **viewport.c**.

B.15.2 Function Documentation

B.15.2.1 int compareViewports (const void * *viewportPtr1*, const void * *viewportPtr2*)

Compares the z position of two Viewports.

Intended for use with qsort(). 0 is returned if the two Viewports have the same zpos; the result is positive if the first is in front of the second, and negative if the second is in front of the first.

Parameters:

← ***viewportPtr1*** A void pointer to the first **Viewport**(p. 45).

← ***viewportPtr2*** A void pointer to the second **Viewport**(p. 45).

Returns:

The difference in the z-position of the input Viewports.

Author:

Nicolas Ward '05

Definition at line 25 of file viewport.c.

References Viewport::zpos.

Referenced by runRune().

B.15.2.2 void freeViewport (Viewport * *viewport*)

Frees a **Viewport**(p. 45) data structure and all of its children

Parameters:

← ***viewport*** The **Viewport**(p. 45) structure to be freed.

Author:

Nicolas Ward '05

Definition at line 42 of file viewport.c.

References freeVisualizer().

Referenced by freeView().

Here is the call graph for this function:

B.15.2.3 bool intersectViewports (Viewport * *viewport1*, Viewport * *viewport2*)

Checks if one **Viewport**(p. 45) overlaps another, to determine if the overlapped **Viewport**(p. 45) needs to be redrawn.

Checks if the second (overlapping) **Viewport**(p. 45) is in front, and if it is somewhere inside the first (overlapped) Viewport's bounding box.

Parameters:

← ***viewport1*** A pointer to the overlapped **Viewport**(p. 45).

← ***viewport2*** A pointer to the overlapping **Viewport**(p. 45).

Returns:

True if there is any overlap.

Author:

Nicolas Ward '05

Definition at line 91 of file viewport.c.

Referenced by updateView().

B.15.2.4 void updateViewport (Viewport * *viewport*, Rune * *rune*)

Updates a Viewport's display with newly visualized data.

The many #ifdefs in this function provide for compile-time hooks that select the drawing method being used. The constants are #defined in **rune.h**(p. 89).

Parameters:

← ***viewport*** The **Viewport**(p. 45) being updated.

← ***rune*** A pointer to the **Rune**(p. 40) data structure.

Author:

Nicolas Ward '05

Todo

Figure out which surface method works best, and then strip out the unused code.

Todo

Optimize!

Definition at line 112 of file viewport.c.

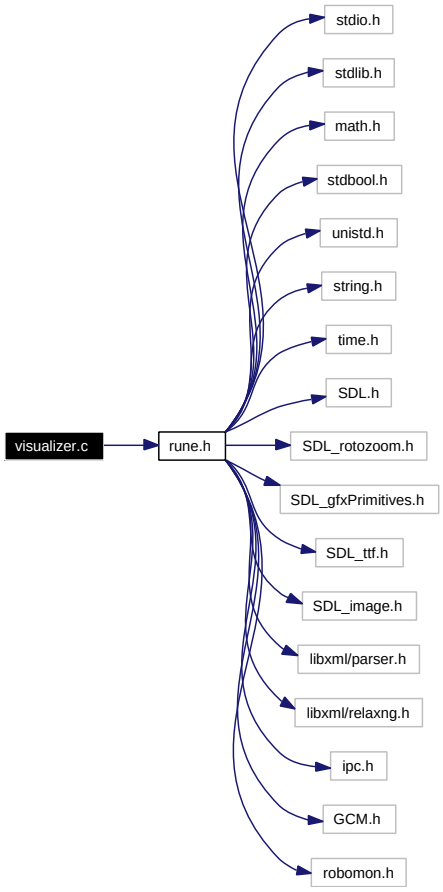
References BMASK, Visualizer::surface, and Viewport::visualizer.

Referenced by updateView().

B.16 visualizer.c File Reference

`#include <rune.h>`

Include dependency graph for visualizer.c:



Defines

- #define **FILLED_PIE** filledpieRGBA

Functions

- void **drawPTZData** (**Visualizer** ∗visualizer, int shift, int size)
- void **freeVisualizer** (**Visualizer** ∗visualizer)
- void **getVisualizerBindings** (char ∗name, **Visualizer** ∗visualizer)
- void **printImage** (GCM_Common_Image ∗image)
- void **printVisualizer** (**Visualizer** ∗visualizer)
- void **resizeImage** (unsigned char ∗input, int inputW, int inputH, int inputD, unsigned char ∗output, int outputW, int outputH, int outputD)

- void **visualizeCameraImage** (**Visualizer** ∗visualizer)
- void **visualizeGroundPlane** (**Visualizer** ∗visualizer)
- void **visualizeMapData** (**Visualizer** ∗visualizer)
- void **visualizeNightMode** (**Visualizer** ∗visualizer)
- void **visualizePanData** (**Visualizer** ∗visualizer)
- void **visualizeRangeData** (**Visualizer** ∗visualizer)
- void **visualizeTiltData** (**Visualizer** ∗visualizer)

B.16.1 Detailed Description

Contains visualizer functions which handle incoming IPC message data and convert that data into visual data which can be rendered onscreen.

Author:
Nicolas Ward ’05

Date:
2005.03.20

Definition in file **visualizer.c**.

B.16.2 Define Documentation

B.16.2.1 #define FILLED_PIE filledpieRGBA

Referenced by visualizeRangeData().

B.16.3 Function Documentation

B.16.3.1 void drawPTZData (Visualizer ∗ *visualizer*, int *shift*, int *size*)

Draws pan and zoom or tilt and zoom data to a **Visualizer**(p. 48) surface.

Parameters:
← ***visualizer*** The pan or tiltVisualizer that called this function.
← ***shift*** The center of the indicator bar, in pixels from one end of the calling Visualizer’s surface.
← ***size*** The size of the indicator bar in pixels.

Author:
Nicolas Ward ’05

Definition at line 22 of file visualizer.c.

References BMASK, GMASK, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by visualizePanData(), and visualizeTiltData().

B.16.3.2 void freeVisualizer (Visualizer * *visualizer*)

Frees a **Visualizer**(p. 48) data structure.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) structure to be freed.

Author:

Nicolas Ward '05

Definition at line 131 of file visualizer.c.

Referenced by freeViewport().

B.16.3.3 void getVisualizerBindings (char * *name*, Visualizer * *visualizer*)

Hashes a **Visualizer**(p. 48) function name to a VisualizerFunction pointer and a string name of an IPC message.

Parameters:

← ***name*** The string name of the function being hashed.

↔ ***visualizer*** The **Visualizer**(p. 48) that will be bound to the returned visualization function and data message.

Author:

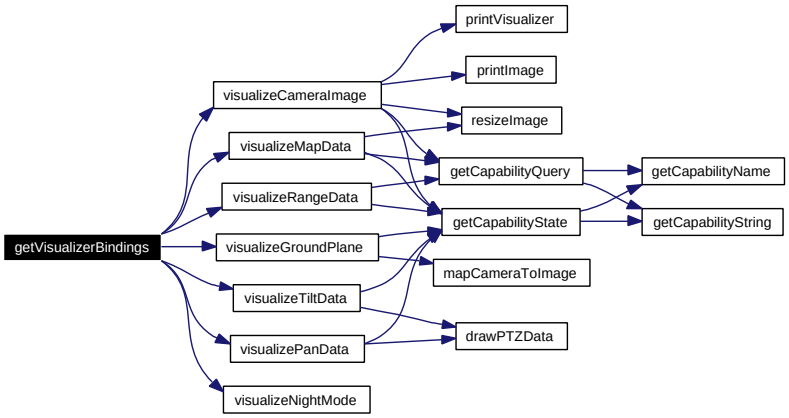
Nicolas Ward '05

Definition at line 159 of file visualizer.c.

References R_TOGGLE_NIGHT_MODE, visualizeCameraImage(), visualizeGroundPlane(), visualizeMapData(), visualizeNightMode(), visualizePanData(), visualizeRangeData(), and visualizeTiltData().

Referenced by parseVisualizer().

Here is the call graph for this function:



B.16.3.4 void printImage (GCM_Common_Image * *image*)

Outputs the text metadata that is stored in a GCM image data structure.

Parameters:

← ***image*** The image whose metadata will be output.

Author:

Nicolas Ward '05

Definition at line 207 of file visualizer.c.

Referenced by visualizeCameraImage().

B.16.3.5 void printVisualizer (Visualizer * *visualizer*)

Outputs the text metadata that is stored in a **Visualizer**(p. 48) data structure.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) whose metadata will be output.

Author:

Nicolas Ward '05

Definition at line 269 of file visualizer.c.

Referenced by visualizeCameraImage().

B.16.3.6 void resizeImage (unsigned char * *input*, int *inputW*, int *inputH*, int *inputD*, unsigned char * *output*, int *outputW*, int *outputH*, int *outputD*)

Resizes an PPM-formatted image by interpolating the unsigned character data of the image.

This function cannot scale an image down because it does not do any sampling. It probably isn't particularly good or efficient at what it does either. Use sparingly.

This function does not handle color depths correctly, but it does work well enough to convert a PGM or PPM to a PPM.

Parameters:

← ***input*** The character data of the input image.

← ***inputW*** The width of the input image in pixels.

← ***inputH*** The height of the input image in pixels.

← ***inputD*** The color depth of the input image in bytes.

→ ***output*** A pre-allocated array where the character data of the output image will be written.

← ***outputW*** The width of the output image in pixels.

← ***outputH*** The height of the output image in pixels.

← ***outputD*** The color depth of the output image in bytes.

Author:
Nicolas Ward ’05

Todo
Fix this function or find a replacement that’s fast.

Definition at line 317 of file visualizer.c.

Referenced by visualizeCameraImage(), and visualizeMapData().

B.16.3.7 void visualizeCameraImage (Visualizer * *visualizer*)

Uses the appropriate GCM RLE decoding function to convert the compressed image data sent over IPC into a raw PPM-formatted color image based on the specified size and quality.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) containing the image data.

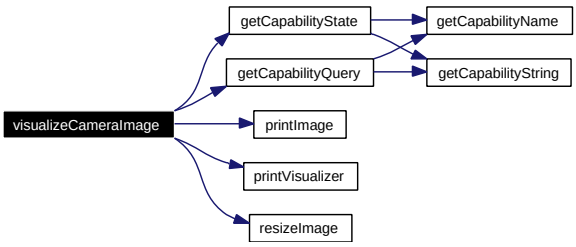
Author:
Nicolas Ward ’05

Definition at line 376 of file visualizer.c.

References AMASK, BMASK, getCapabilityQuery(), getCapabilityState(), GMASK, ImageRequest::handled, printImage(), printVisualizer(), resizeImage(), and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.16.3.8 void visualizeGroundPlane (Visualizer * *visualizer*)

Uses incoming camera state data sent over IPC to calculate and draw indicators that approximate a ground plane in the camera’s view.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

Author:

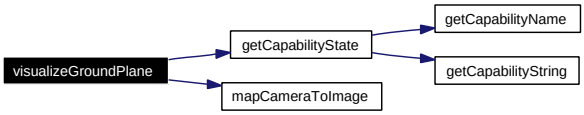
Nicolas Ward '05

Definition at line 566 of file visualizer.c.

References BMASK, getCapabilityState(), GMASK, mapCameraToImage(), R_SDL_-SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.16.3.9 void visualizeMapData (Visualizer * *visualizer*)

Uses the appropriate GCM RLE decoding function to convert the compressed map data sent over IPC into a raw PPM-formatted color image based on the mapping of different types of interest points.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the map data.

Author:

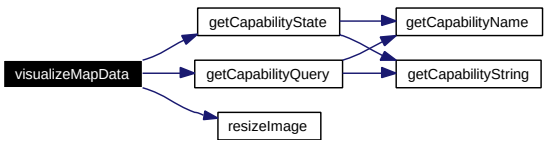
Nicolas Ward '05

Definition at line 672 of file visualizer.c.

References AMASK, BMASK, getCapabilityQuery(), getCapabilityState(), GMASK, Common-Request::handled, resizeImage(), and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.16.3.10 void visualizeNightMode (Visualizer * *visualizer*)

Responds to a local R_TOGGLE_NIGHT_MODE message to display an icon as needed.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) that will draw the icon.

Author:

Nicolas Ward '05

Definition at line 879 of file visualizer.c.

References BMASK, GMASK, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

B.16.3.11 void visualizePanData (Visualizer * *visualizer*)

Draws a bar indicating current pan position relative to the center line, as well as the current zoom setting in terms of angular field-of-view.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

Author:

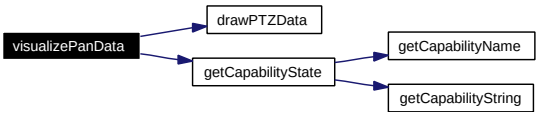
Nicolas Ward '05

Definition at line 962 of file visualizer.c.

References drawPTZData(), and getCapabilityState().

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.16.3.12 void visualizeRangeData (Visualizer * *visualizer*)

Draws a range data representation. Assumes a circular robot with radial sensors; in other words, a Magellan.

Parameters:

← ***visualizer*** The **Visualizer**(p. 48) containing the robot state data.

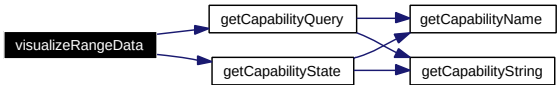
Author:
Nicolas Ward ’05

Definition at line 1021 of file visualizer.c.

References BMASK, FILLED_PIE, getCapabilityQuery(), getCapabilityState(), GMASK, CommonRequest::handled, R_SDL_SURFACE_FLAGS, and RMASK.

Referenced by getVisualizerBindings().

Here is the call graph for this function:



B.16.3.13 void visualizeTiltData (Visualizer * *visualizer*)

Draws a bar indicating current tilt position relative to the center line, as well as the current zoom setting in terms of angular field-of-view.

Parameters:
← ***visualizer*** The **Visualizer**(p. 48) containing the camera state data.

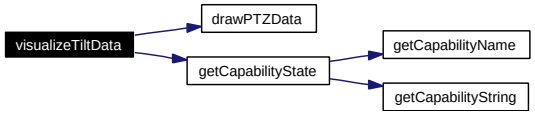
Author:
Nicolas Ward ’05

Definition at line 1222 of file visualizer.c.

References drawPTZData(), and getCapabilityState().

Referenced by getVisualizerBindings().

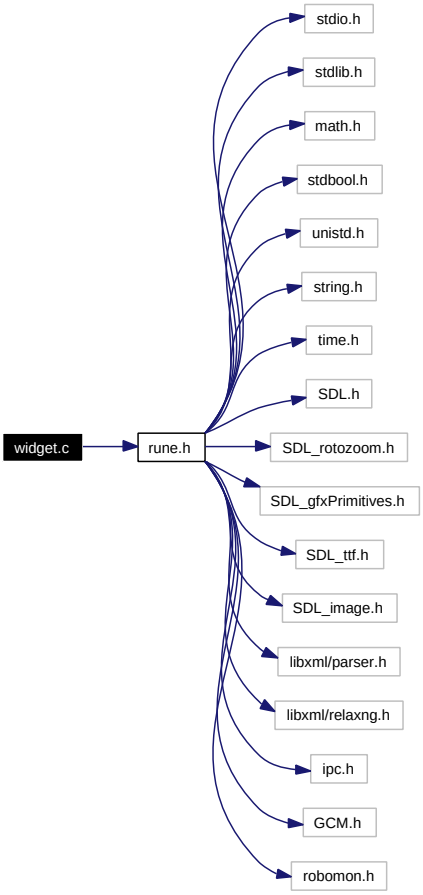
Here is the call graph for this function:



B.17 widget.c File Reference

`#include <rune.h>`

Include dependency graph for widget.c:



Functions

- **WidgetHandler getWidgetHandler** (char ∗name)
- void **mapImageToCamera** (double imageSpaceX, double imageSpaceY, double imageSpaceW, double imageSpaceH, GCM_CameraState ∗camera, double ∗cameraSpaceX, double ∗cameraSpaceY)
- void **mapImageToWorld** (double imageSpaceX, double imageSpaceY, double imageSpaceW, double imageSpaceH, **Robot** ∗robot, double ∗worldSpaceX, double ∗worldSpaceY)
- void **mapCameraToImage** (double cameraSpaceX, double cameraSpaceY, GCM_CameraState ∗camera, double imageSpaceW, double imageSpaceH, double ∗imageSpaceX, double ∗imageSpaceY)
- void **mapWorldToImage** (double worldSpaceX, double worldSpaceY, **Robot** ∗robot, double imageSpaceW, double imageSpaceH, double ∗imageSpaceX, double ∗imageSpaceY)
- void **widgetAdjustPan** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetAdjustPanTilt** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetAdjustTilt** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetAdjustZoom** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetCorrectLandmark** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)

- void **widgetHomePTZ** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetQuit** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetImageRequest** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetLandmark** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetVictim** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetSetSpeed** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)
- void **widgetToggleNightMode** (**Widget** ∗widget, **Event** ∗event, SDL_Event ∗sdlEvent)

B.17.1 Detailed Description

Contains **Widget**(p. 51) functions which handle SDL events and respond to them by modifying Rune's state appropriately.

Author:
Nicolas Ward '05

Date:
2005.03.20

Definition in file **widget.c**.

B.17.2 Function Documentation

B.17.2.1 WidgetHandler getWidgetHandler (char ∗ *name*)

Hashes a **Widget**(p. 51) handler function name to a WidgetHandler pointer.

Parameters:
← ***name*** The string name of the function being hashed.

Returns:
The function pointer.

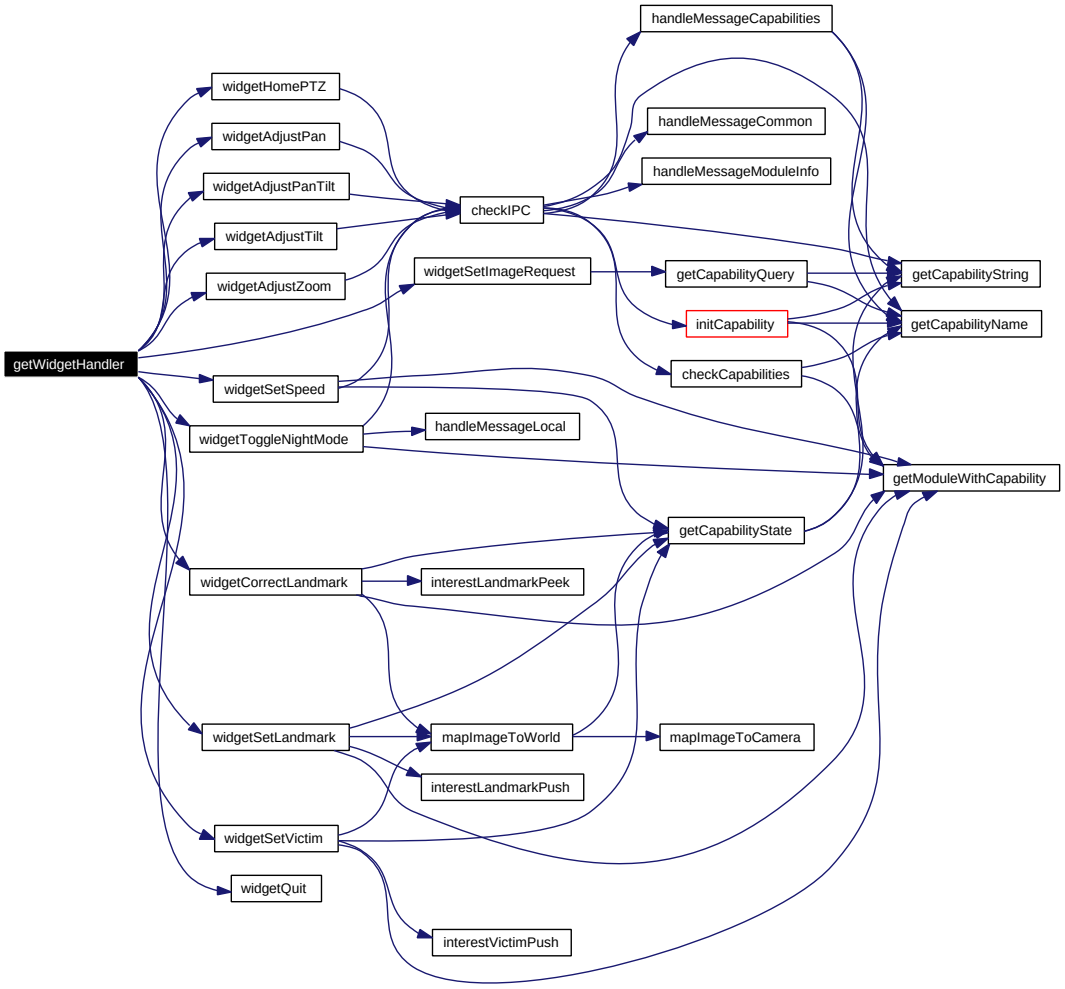
Author:
Nicolas Ward '05

Definition at line 20 of file widget.c.

References widgetAdjustPan(), widgetAdjustPanTilt(), widgetAdjustTilt(), widgetAdjustZoom(), widgetCorrectLandmark(), widgetHomePTZ(), widgetQuit(), widgetSetImageRequest(), widgetSetLandmark(), widgetSetSpeed(), widgetSetVictim(), and widgetToggleNightMode().

Referenced by parseWidget().

Here is the call graph for this function:



B.17.2.2 void mapCameraToImage (double *cameraSpaceX*, double *cameraSpaceY*, GCM_CameraState * *camera*, double *imageSpaceW*, double *imageSpaceH*, double * *imageSpaceX*, double * *imageSpaceY*)

Performs a coordinate mapping from an (x,y) position relative to the robot's axes to an (x,y) pixel value in an image.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

→ ***cameraSpaceX*** The x-coordinate in camera space in meters.

→ ***cameraSpaceY*** The y-coordinate in camera space in meters.

← ***camera*** The camera whose coordinate space is being used.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

→ ***imageSpaceX*** The x-coordinate in image space in pixels.

→ ***imageSpaceY*** The y-coordinate in image space in pixels.

Author:

Nicolas Ward '05

```
*
* Convert from 2-D cartesian in camera space to 3-D polar in camera space
*  Angles /_hc and /_vc are measured in radians from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*  Ranges Rxy, and Ryz are measured in meters from the camera mountpoint
*    cameraSpaceRadiusH is in the ground plane
*    cameraSpaceRadiusV is in the vertical plane through the robot
*  Point (x,y) is measured in meters from below the camera's center point
*    cameraSpaceX is positive right
*    cameraSpaceY is positive forwards
*  Camera height is measured in meters from the ground plane
*    height is positive up
*
*    (top view)                        (side view)
*             . world point          camera ._______
*         |  /                              |\) -cameraSpaceAngleV
*        y|~/ Rxy                        h | \ Ryz
*         |/__    ~cameraSpaceAngleH       |  \
*         ' x                               |__(\
*  camera                                       'world point
*  center                      by Alternate Interior Angle theorem
*
*


*
* Convert from 3-D polar in camera space to radians in image space
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  Angles /_p and /_t are measured to the center of the camera's image
*    pan is positive left (CCW from straight ahead)
*    tilt is positive up (CCW from downwards vertical)
*  Angles /_hc and /_vc are measured from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*
*  camera .            \   |
*         |\            \  |
*         |~\ /_t    /_p |
*         |  \           \|
*         |   \           'camera
*      (side view)     (top view)
*
*
```

```
*
* Convert from pixels to radians in image space
*  Pixel (x,y) is measured from the upper left of the image
*    imageSpaceX is positive right
*    imageSpaceY is positive down
*  The pixel dimensions of the image are imageSpaceW by imageSpaceH
*    The image center is at (w/2,h/2)
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  The angular dimensions of the image are HFOV by VFOV
*
*     (side view)      (top view)            (front view)
*         /|  \           /|  \                _________________
*        / |  |          / |  |              |  x              | \
*  VFOV /  |  |    HFOV /  |  |             y|        | /_v    | |
*      <)  |  }h       <)  |  }w             |     ___|        | }h
*       \  |  |         \  |  |              |    /_h 'center  | |
*        \ |  |          \ |  |              |                 | |
*         \|  /           \|  /              |_________________| /
* image plane     image plane
*                                            \________w________/
*
*
```

Definition at line 283 of file widget.c.

Referenced by mapWorldToImage(), and visualizeGroundPlane().

B.17.2.3 void mapImageToCamera (double *imageSpaceX*, double *imageSpaceY*, double *imageSpaceW*, double *imageSpaceH*, GCM_CameraState * *camera*, double * *cameraSpaceX*, double * *cameraSpaceY*)

Performs a coordinate mapping from an (x,y) pixel value in an image to an (x,y) position value relative to the ground plane projection of the camera's axes.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

← ***imageSpaceX*** The x-coordinate in image space in pixels.

← ***imageSpaceY*** The y-coordinate in image space in pixels.

← ***imageSpaceW*** The width of the image space in pixels.

← ***imageSpaceH*** The height of the image space in pixels.

← ***camera*** The camera whose coordinate space is being used.

→ ***cameraSpaceX*** The x-coordinate in camera space in meters.

→ ***cameraSpaceY*** The y-coordinate in camera space in meters.

Author:
Nicolas Ward ’05

```
*
* Convert from pixels to radians in image space
*  Pixel (x,y) is measured from the upper left of the image
*    imageSpaceX is positive right
*    imageSpaceY is positive down
*  The pixel dimensions of the image are imageSpaceW by imageSpaceH
*    The image center is at (w/2,h/2)
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  The angular dimensions of the image are HFOV by VFOV
*
*     (side view)      (top view)            (front view)
*         /|  \           /|  \                _________________
*        / |  |          / |  |              |  x              | \
*  VFOV /  |  |    HFOV /  |  |            y|        | /_v    | |
*      <)  |  }h       <)  |  }w            |     ___|        | }h
*       \  |  |         \  |  |             |    /_h ‘center  | |
*        \ |  |          \ |  |             |                 | |
*         \|  /           \|  /             |_________________| /
* image plane     image plane
*                                           \________w________/
*
*


*
* Convert from radians in image space to 3-D polar in camera space
*  Angles /_h and /_v are measured from the image center
*    imageSpaceH is positive left in image
*    imageSpaceV is positive up in image
*  Angles /_p and /_t are measured to the center of the camera’s image
*    pan is positive left (CCW from straight ahead)
*    tilt is positive up (CCW from downwards vertical)
*  Angles /_hc and /_vc are measured from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*
*  camera .            \   |
*         |\            \  |
*         |~\ /_t    /_p |
*         |  \           \|
*         |   \           ‘camera
*      (side view)     (top view)
*
*


*
* Convert from 3-D polar in camera space to 2-D cartesian in camera space
*  Angles /_hc and /_vc are measured in radians from the camera mountpoint
*    cameraSpaceAngleH is positive left (CCW from straight ahead)
```

```
*    cameraSpaceAngleV is positive up (CCW from straight ahead)
*  Ranges Rxy, and Ryz are measured in meters from the camera mountpoint
*    cameraSpaceRadiusH is in the ground plane
*    cameraSpaceRadiusV is in the vertical plane through the robot
*  Point (x,y) is measured in meters from below the camera's center point
*    cameraSpaceX is positive right
*    cameraSpaceY is positive forwards
*  Camera height is measured in meters from the ground plane
*    height is positive up
*
*    (top view)                        (side view)
*             . world point          camera ._______
*         |  /                              |\) -cameraSpaceAngleV
*        y|~/ Rxy                         h | \ Ryz
*         |/__    ~cameraSpaceAngleH        |  \
*         ' x                               |__(\
*  camera                                        'world point
*  center                       by Alternate Interior Angle theorem
*
*
```

Definition at line 82 of file widget.c.

Referenced by mapImageToWorld().

B.17.2.4 void mapImageToWorld (double *imageSpaceX*, double *imageSpaceY*, double *imageSpaceW*, double *imageSpaceH*, Robot * *robot*, double * *worldSpaceX*, double * *worldSpaceY*)

Performs a coordinate mapping from an (x,y) pixel value in an image to an (x,y) position value on the global ground plane.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

- ← ***imageSpaceX*** The x-coordinate in image space in pixels.
- ← ***imageSpaceY*** The y-coordinate in image space in pixels.
- ← ***imageSpaceW*** The width of the image space in pixels.
- ← ***imageSpaceH*** The height of the image space in pixels.
- ← ***robot*** The robot whose coordinate space is being used.
- → ***worldSpaceX*** The x-coordinate in world space in meters.
- → ***worldSpaceY*** The y-coordinate in world space in meters.

Author:

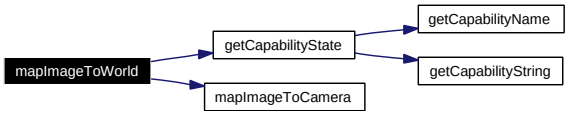
Nicolas Ward '05

Definition at line 208 of file widget.c.

References getCapabilityState(), and mapImageToCamera().

Referenced by widgetCorrectLandmark(), widgetSetLandmark(), and widgetSetVictim().

Here is the call graph for this function:



B.17.2.5 void mapWorldToImage (double *worldSpaceX*, double *worldSpaceY*, Robot * *robot*, double *imageSpaceW*, double *imageSpaceH*, double * *imageSpaceX*, double * *imageSpaceY*)

Performs a coordinate mapping from an (x,y) position value on the global ground plane to an (x,y) pixel value in an image.

See the detailed comments and schematics associated with each mathematical step of this coordinate mapping.

Parameters:

- ← ***worldSpaceX*** The x-coordinate in world space in meters.
- ← ***worldSpaceY*** The y-coordinate in world space in meters.
- ← ***robot*** The robot whose coordinate space is being used.
- ← ***imageSpaceW*** The width of the image space in pixels.
- ← ***imageSpaceH*** The height of the image space in pixels.
- → ***imageSpaceX*** The x-coordinate in image space in pixels.
- → ***imageSpaceY*** The y-coordinate in image space in pixels.

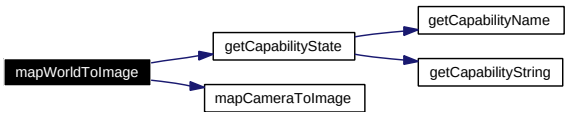
Author:

Nicolas Ward ’05

Definition at line 407 of file widget.c.

References getCapabilityState(), and mapCameraToImage().

Here is the call graph for this function:



B.17.2.6 void widgetAdjustPan (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the pan state of the robot's main camera based on input from a button or key.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

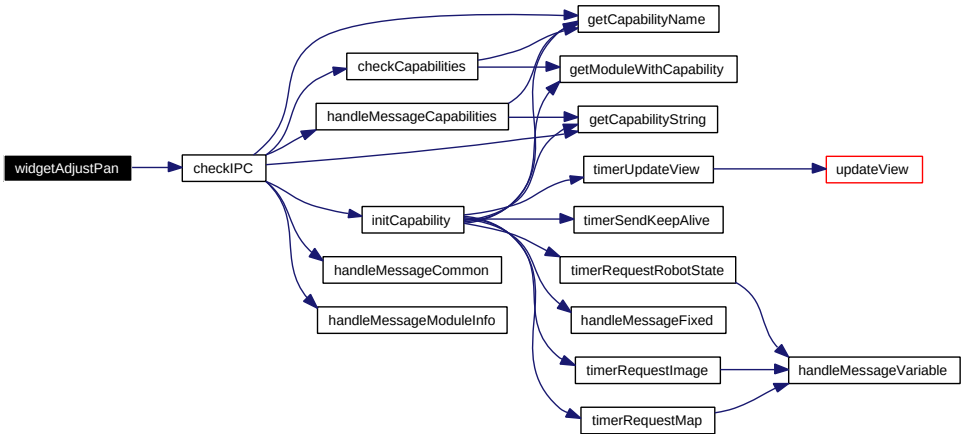
Nicolas Ward '05

Definition at line 481 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.7 void widgetAdjustPanTilt (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the pan-tilt state of the robot's main camera based on input from a hat switch.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

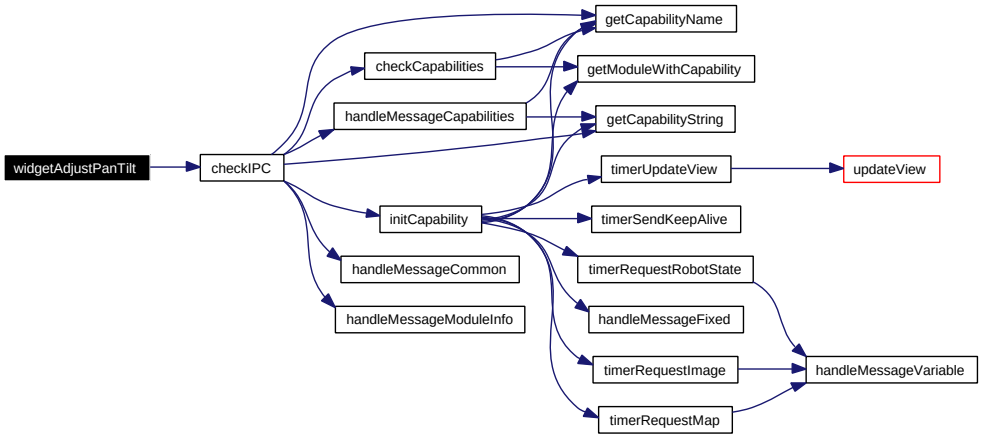
Nicolas Ward '05

Definition at line 531 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.8 void widgetAdjustTilt (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the tilt state of the robot's main camera based on input from a button or key.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

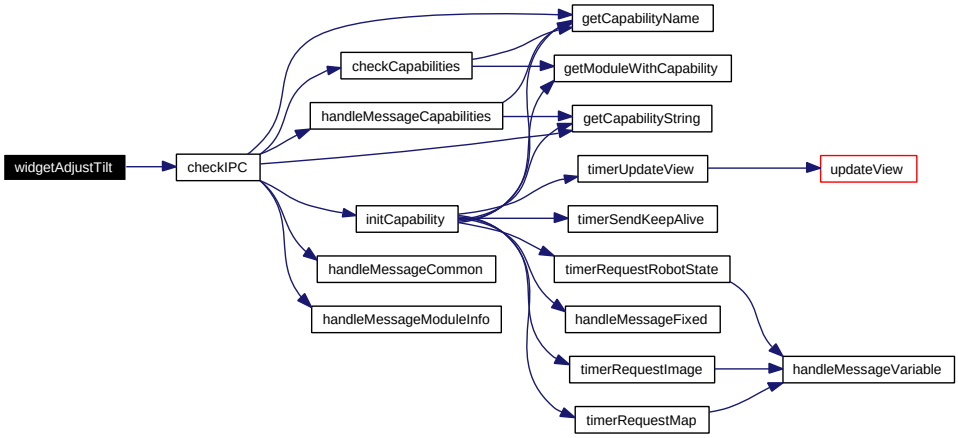
Nicolas Ward '05

Definition at line 585 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.9 void widgetAdjustZoom (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Adjusts the zoom state of the robot's main camera based on input from a button or key.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

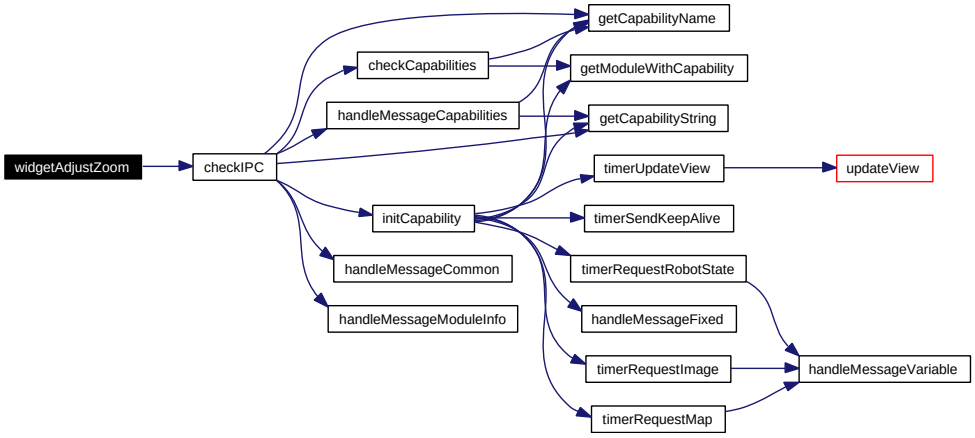
Nicolas Ward '05

Definition at line 635 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.10 void widgetCorrectLandmark (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Performs an odometry correction based on the current landmark.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

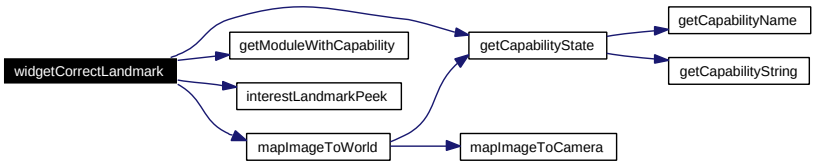
Nicolas Ward '05

Definition at line 684 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestLandmarkPeek(), mapImageToWorld(), InterestPoints::nLandmarks, Viewport::robot, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.11 void widgetHomePTZ (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Homes the PTZ state of the robot's main camera based on any input.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

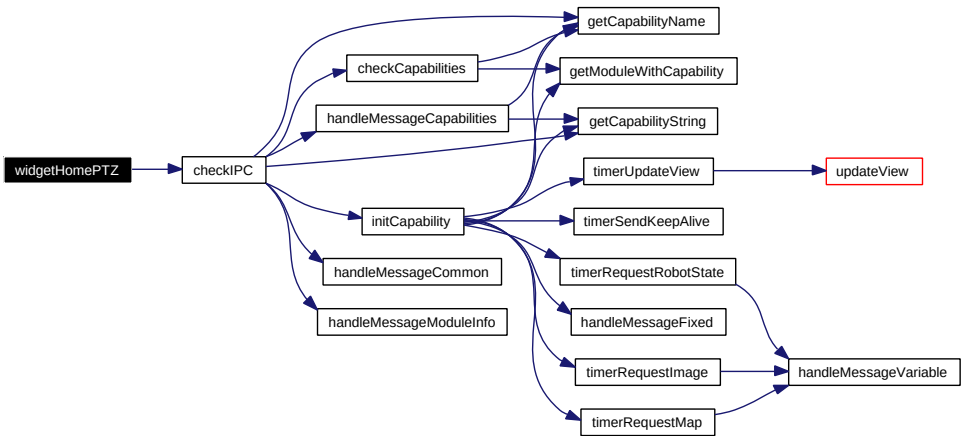
Nicolas Ward '05

Definition at line 806 of file widget.c.

References checkIPC().

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.12 void widgetQuit (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Pushes an SDL_QUIT event onto the queue.

The actual SDL_QUIT event is handled in handleEvent, which calls the quit function, cleans up, and actually quits **Rune**(p. 40).

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

Nicolas Ward '05

Definition at line 838 of file widget.c.

References InterestPoint::type.

Referenced by getWidgetHandler().

B.17.2.13 void widgetSetImageRequest (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Changes the type, size, and quality of the image being requested.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

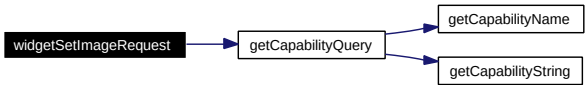
Author:
Nicolas Ward '05

Definition at line 857 of file widget.c.

References getCapabilityQuery(), Control::invert, Control::max, Control::min, test, and Control::type.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.14 void widgetSetLandmark (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Stores a landmark for future use.

Parameters:
← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

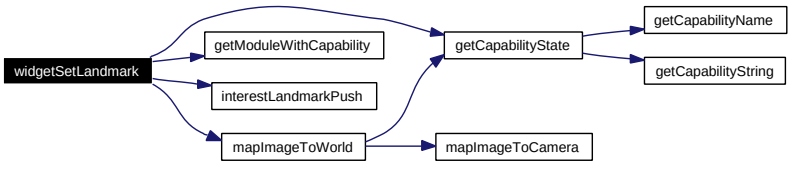
Author:
Nicolas Ward '05

Definition at line 935 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestLandmarkPush(), mapImageToWorld(), InterestPoints::nLandmarks, Viewport::robot, InterestPoint::type, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.15 void widgetSetSpeed (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Sets rotation and translation speeds for the robot, based on joystick axes.

Normally has the functionality to home the robot's PTZ camera, but that is disabled until we integrate some SVM messages into GCM.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

Nicolas Ward '05

Todo

Remove dual-axis dependency check.

Todo

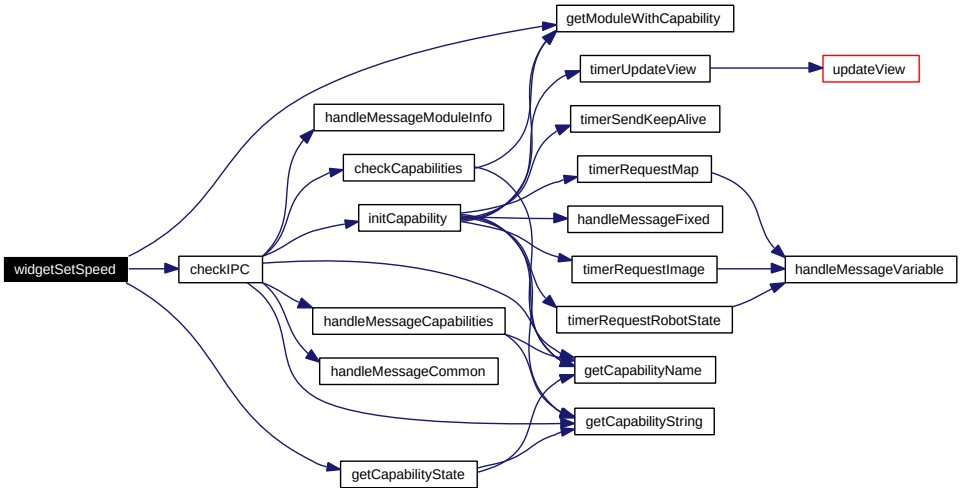
Remove or fix the automatic camera reset.

Definition at line 1162 of file widget.c.

References checkIPC(), Robot::context, getCapabilityState(), getModuleWithCapability(), Control::invert, Control::max, Control::min, Control::name, Viewport::robot, and test.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.16 void widgetSetVictim (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Stores a victim for future use.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

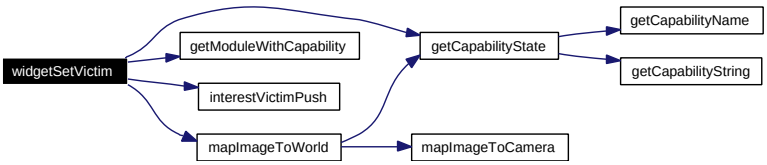
Nicolas Ward '05

Definition at line 1042 of file widget.c.

References Robot::context, getCapabilityState(), getModuleWithCapability(), interestVictimPush(), mapImageToWorld(), InterestPoints::nVictims, Viewport::robot, InterestPoint::type, InterestPoint::x, Viewport::xpos, Viewport::xsize, InterestPoint::y, Viewport::ypos, and Viewport::ysize.

Referenced by getWidgetHandler().

Here is the call graph for this function:



B.17.2.17 void widgetToggleNightMode (Widget * *widget*, Event * *event*, SDL_Event * *sdlEvent*)

Turns the night mode of a camera on or off.

Parameters:

← ***widget*** This WidgetHandler's parent **Widget**(p. 51).

← ***event*** The configured event that was triggered.

← ***sdlEvent*** The SDL_Event that triggered this handler.

Author:

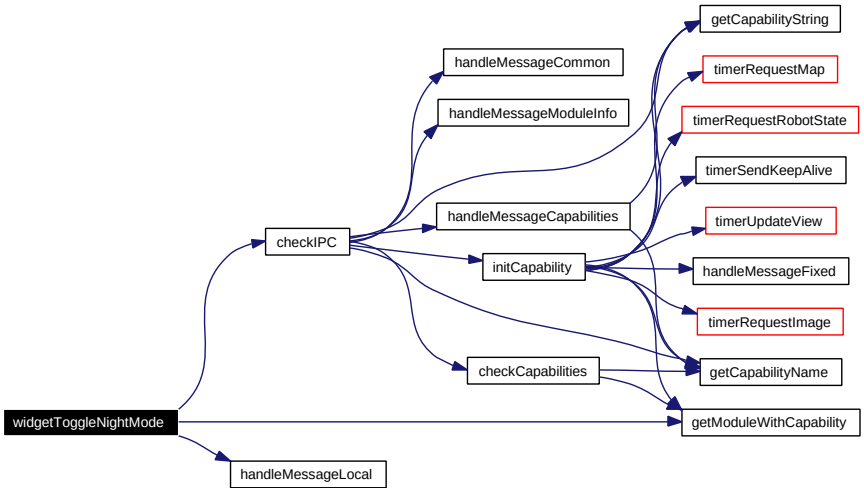
Nicolas Ward '05

Definition at line 1308 of file widget.c.

References checkIPC(), Robot::context, getModuleWithCapability(), handleMessageLocal(), R_-TOGGLE_NIGHT_MODE, and Viewport::robot.

Referenced by getWidgetHandler().

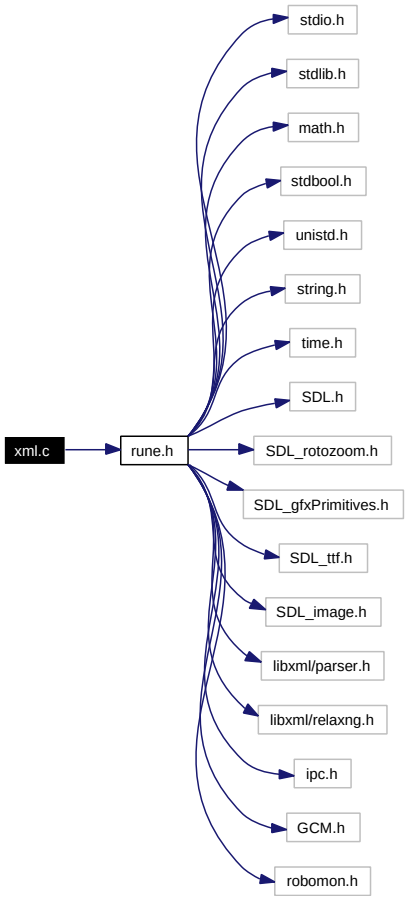
Here is the call graph for this function:



B.18 xml.c File Reference

`#include <rune.h>`

Include dependency graph for xml.c:



Functions

- void **parseControl** (**Control** ∗∗controlPtr, xmlDocPtr doc, xmlNodePtr node)
- void **parseDocument** (char ∗filename, **Rune** ∗rune)
- void **parseEvent** (**Event** ∗∗eventPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseFont** (**Font** ∗font, xmlDocPtr doc, xmlNodePtr node)
- void **parseJoystick** (**Joystick** ∗∗joystickPtr, xmlDocPtr doc, xmlNodePtr node)
- void **parseRobot** (**Robot** ∗∗robotPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseView** (**View** ∗∗viewPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseViewport** (**Viewport** ∗∗viewportPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseVisualizer** (**Visualizer** ∗∗visualizerPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **parseWidget** (**Widget** ∗∗widgetPtr, **Rune** ∗rune, xmlDocPtr doc, xmlNodePtr node)
- void **validateDocument** (char ∗filename, xmlDocPtr doc)

B.18.1 Detailed Description

Contains functions for reading, validating, allocating, and loading **Rune**(p. 40) data structures from a parsed XML configuraiton file.

Author:
Nicolas Ward '05

Date:
2005.03.19

Definition in file **xml.c**.

B.18.2 Function Documentation

B.18.2.1 void parseControl (Control ∗∗ *controlPtr*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <control> element from an XML configuration file.

Parameters:
→ ***controlPtr*** A pointer to the **Control**(p. 26) data structure being allocated by this function.
← ***doc*** A pointer to the XML document being read.
← ***node*** A pointer to the current XML node.

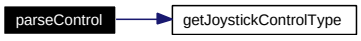
Author:
Nicolas Ward '05

Definition at line 22 of file xml.c.

References Control::calibrate, getJoystickControlType(), Control::index, Control::invert, JOYSTICK_CONTROL_AXIS, Control::max, Control::min, Control::name, and Control::type.

Referenced by parseJoystick().

Here is the call graph for this function:



B.18.2.2 void parseDocument (char ∗ *filename*, Rune ∗ *rune*)

Parses an XML document, validates it, and loads the **Rune**(p. 40) data structure.

Parameters:
← ***filename*** The filename of the XML configuration file.

↔ ***rune*** The **Rune**(p. 40) data structure.

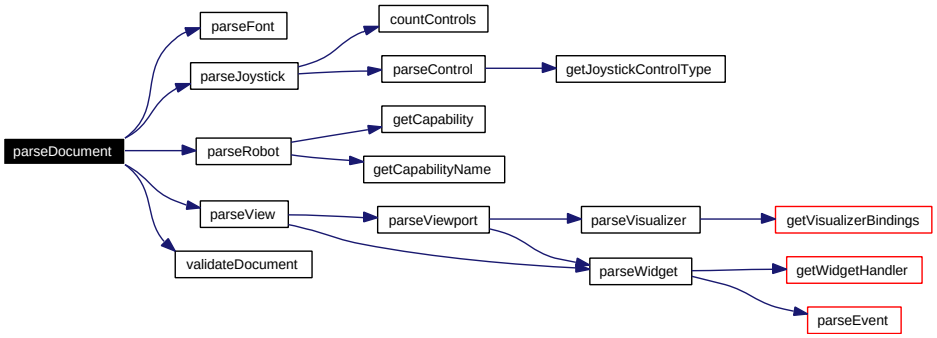
Author:
Nicolas Ward '05

Definition at line 168 of file xml.c.

References parseFont(), parseJoystick(), parseRobot(), parseView(), and validateDocument().

Referenced by main(), and runRune().

Here is the call graph for this function:



B.18.2.3 void parseEvent (Event ∗∗ *eventPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses an <event> element from an XML configuration file.

Parameters:

→ ***eventPtr*** A pointer to the **Event**(p. 28) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

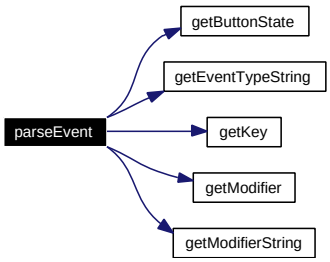
Author:
Nicolas Ward '05

Definition at line 319 of file xml.c.

References Joystick::config, Event::control, Joystick::controls, Event::event, getButtonState(), getEventTypeString(), getKey(), getModifier(), getModifierString(), Control::index, Event::joystick, JOYSTICK_CONTROL_AXIS, JOYSTICK_CONTROL_BALL, JOYSTICK_CONTROL_BUTTON, JOYSTICK_CONTROL_HAT_SWITCH, Control::name, Joystick::nControls, Event::options, and Control::type.

Referenced by parseWidget().

Here is the call graph for this function:



B.18.2.4 void parseFont (Font * *font*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <font> element from an XML state file.

Parameters:

→ ***font*** A pointer to the **Font**(p. 30) data structure being allocated by this function.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

Nicolas Ward '05

Definition at line 694 of file xml.c.

Referenced by parseDocument().

B.18.2.5 void parseJoystick (Joystick ** *joystickPtr*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <joystick> element from an XML configuration file.

Parameters:

→ ***joystickPtr*** A pointer to the **Joystick**(p. 35) data structure being allocated by this function.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

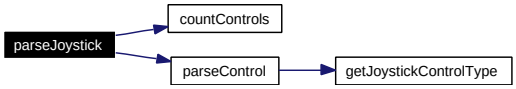
Nicolas Ward '05

Definition at line 742 of file xml.c.

References Joystick::config, Joystick::controls, countControls(), Control::joystick, Joystick::joystick, Joystick::nControls, and parseControl().

Referenced by parseDocument().

Here is the call graph for this function:



B.18.2.6 void parseRobot (Robot ∗∗ *robotPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <robot> element from an XML configuration file.

Parameters:

→ ***robotPtr*** A pointer to the **Robot**(p. 37) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

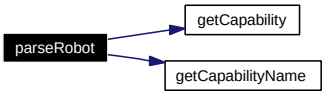
Nicolas Ward ’05

Definition at line 820 of file xml.c.

References Capability::cap, getCapability(), getCapabilityName(), Robot::haveCaps, Robot::hostname, Robot::moduleInfo, Robot::name, Capability::query, Capability::ready, Capability::robot, Capability::state, and Robot::wantCaps.

Referenced by parseDocument().

Here is the call graph for this function:



B.18.2.7 void parseView (View ∗∗ *viewPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses the <view> element from an XML configuration file.

Parameters:

→ ***viewPtr*** A pointer to the **View**(p. 42) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

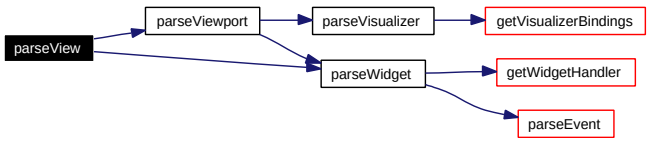
Author:
Nicolas Ward '05

Definition at line 935 of file xml.c.

References View::fullscreen, View::nViewports, View::nWidgets, parseViewport(), parseWidget(), Widget::view, Viewport::view, View::viewports, View::widgets, View::xsize, and View::ysize.

Referenced by parseDocument().

Here is the call graph for this function:



B.18.2.8 void parseViewport (Viewport ∗∗ *viewportPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <viewport> element from an XML configuration file.

Parameters:

→ ***viewportPtr*** A pointer to the **Viewport**(p. 45) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

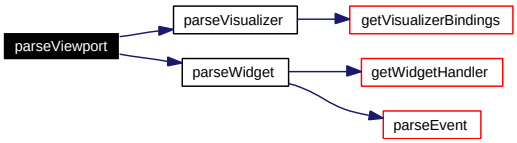
Author:
Nicolas Ward '05

Definition at line 1084 of file xml.c.

References Viewport::nWidgets, parseVisualizer(), parseWidget(), Viewport::robot, Viewport::transparency, Viewport::view, Visualizer::viewport, Widget::viewport, Viewport::visible, Viewport::visualizer, Viewport::widgets, Viewport::xpos, View::xsize, Viewport::xsize, Viewport::ypos, View::ysize, Viewport::ysize, and Viewport::zpos.

Referenced by parseView().

Here is the call graph for this function:



B.18.2.9 void parseVisualizer (Visualizer ∗∗ *visualizerPtr*, Rune ∗ *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <visualizer> element from an XML configuration file.

Parameters:

→ ***visualizerPtr*** A pointer to the **Visualizer**(p. 48) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

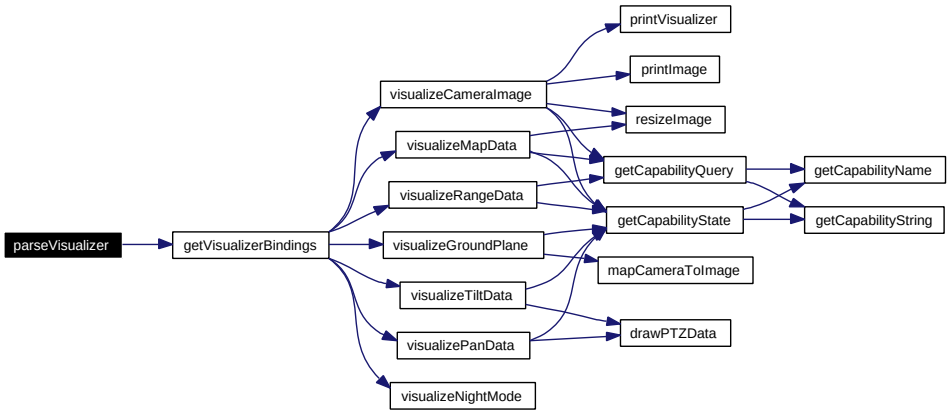
Nicolas Ward '05

Definition at line 1336 of file xml.c.

References Visualizer::data, Visualizer::function, getVisualizerBindings(), Visualizer::option, Visualizer::surface, Visualizer::viewport, Visualizer::xsize, and Visualizer::ysize.

Referenced by parseViewport().

Here is the call graph for this function:



B.18.2.10 void parseWidget (Widget ** *widgetPtr*, Rune * *rune*, xmlDocPtr *doc*, xmlNodePtr *node*)

Parses a <widget> element from an XML configuration file.

Parameters:

→ ***widgetPtr*** A pointer to the **Widget**(p. 51) data structure being allocated by this function.

↔ ***rune*** The **Rune**(p. 40) data structure.

← ***doc*** A pointer to the XML document being read.

← ***node*** A pointer to the current XML node.

Author:

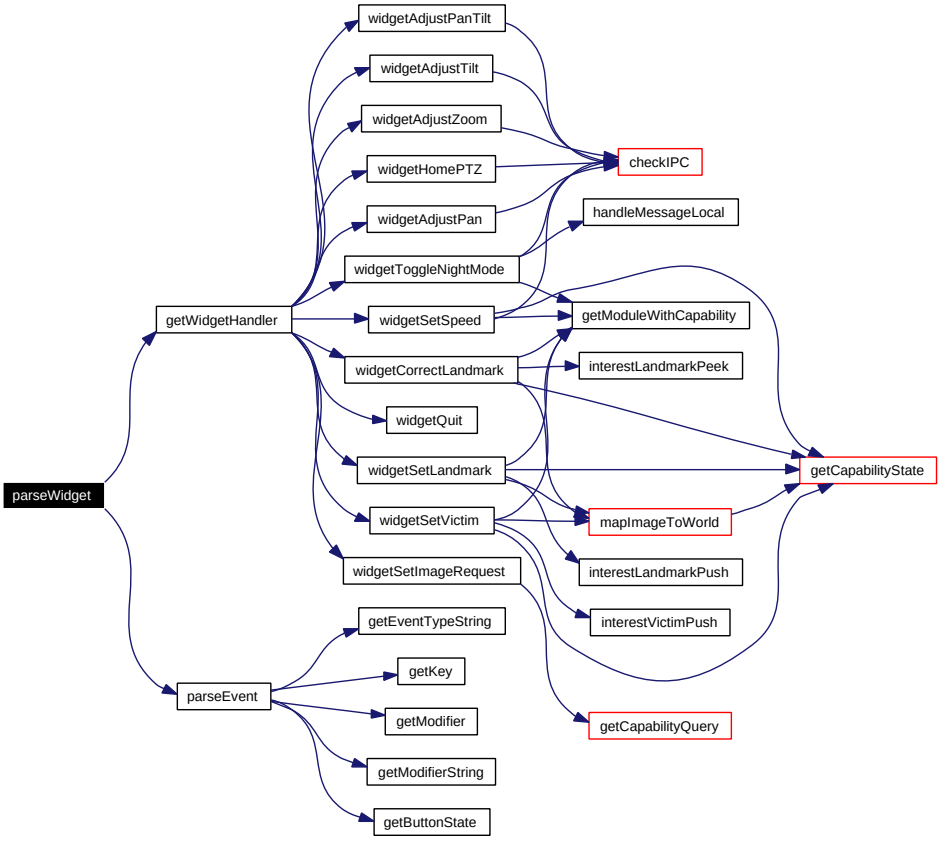
Nicolas Ward '05

Definition at line 1451 of file xml.c.

References Widget::events, getWidgetHandler(), Widget::handler, Widget::history, Widget::nEvents, parseEvent(), Widget::view, and Widget::viewport.

Referenced by parseView(), and parseViewport().

Here is the call graph for this function:



B.18.2.11 void validateDocument (char * *filename*, xmlDocPtr *doc*)

Validates an XML document using a parsed XML schema.

Parameters:

← ***filename*** The filename of the XML schema file.

← ***doc*** A pointer to the XML document being read.

Author:

Nicolas Ward '05

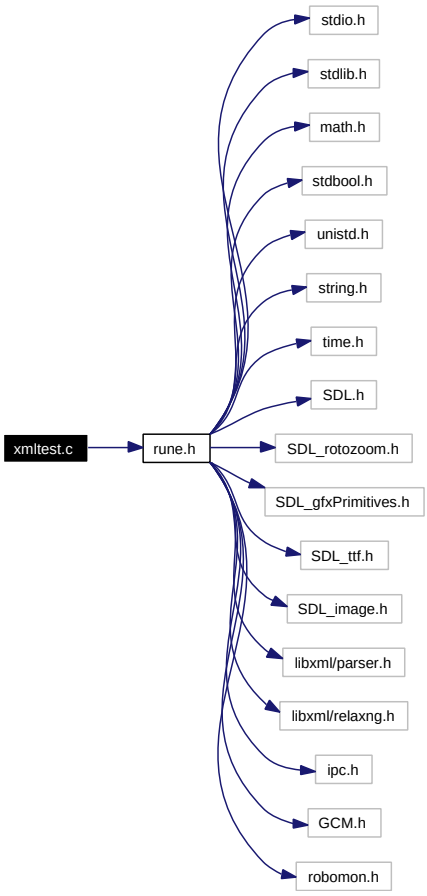
Definition at line 1542 of file xml.c.

Referenced by parseDocument().

B.19 xmltest.c File Reference

`#include <rune.h>`

Include dependency graph for xmltest.c:



Functions

- int **main** (int argc, char **argv)

B.19.1 Detailed Description

Contains a simple main loop to parse a **Rune**(p. 40) XML configuration file.

Author:
Nicolas Ward '05

Date:
2005.03.20

Definition in file **xmltest.c**.

B.19.2 Function Documentation

B.19.2.1 int main (int *argc*, char ** *argv*)

The main joystick querying function.

Checks command line arguments, allocates the **Rune**(p. 40) state data structures, and parses the XML configuration file.

Parameters:
← ***argc*** The number of command line arguments.
← ***argv*** The array of command line argument strings.

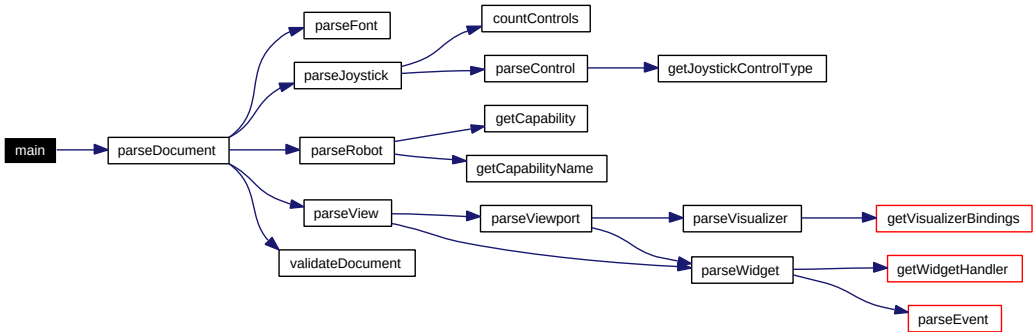
Returns:
0 on successful execution, -1 on error.

Author:
Nicolas Ward '05

Definition at line 23 of file xmltest.c.

References parseDocument().

Here is the call graph for this function:


main
parseDocument
parseFont
parseJoystick
countControls
parseControl
getJoystickControlType
parseRobot
getCapability
getCapabilityName
parseView
parseViewport
parseVisualizer
getVisualizerBindings
validateDocument
parseWidget
getWidgetHandler
parseEvent

Index

-H-

-S-

-W-

-X-

-Y-

-Z-