ANALYSIS TS2PHC SOURCE CODE

6. TS2PHC

Precision Time Protocol (PTP) is a protocol used to synchronize clocks in a network. It is defined in the IEEE 1588 standard. PTP enables sub-microsecond accuracy in clock synchronization, making it a popular choice in applications such as industrial automation, telecommunications, and financial trading.

A hardware clock (PHC) is a clock that is designed to provide precise and accurate timekeeping. PHCs are typically used in embedded systems and networking devices to synchronize clocks across a network.

When ts2phc is started, it reads the configuration file to determine the settings for the PTP and PPS sources. The configuration file specifies the IP address or network interface of the PTP source, as well as the PPS source device or interface.

The daemon then initializes the PTP clock and the PPS source. The PTP clock is set to operate in slave mode, which means that it synchronizes its clock to an external PTP clock source. The PPS source is set up to generate a pulse signal at regular intervals, typically once per second.

Once the PTP and PPS sources are initialized, ts2phc sets up event listeners to receive time stamp and pulse signals. When a time stamp is received from the PTP source, ts2phc uses it to adjust the system clock to match the PTP clock. When a pulse signal is received from the PPS source, ts2phc uses it to adjust the system clock to match the PPS signal.

To ensure accurate timekeeping, ts2phc employs various techniques for filtering and smoothing the time stamp and pulse signals. The daemon also applies various correction values, such as delay and offset corrections, to compensate for any time drift or latency in the PTP and PPS sources.

In addition to synchronizing the system clock, ts2phc provides various diagnostic and debugging features. The daemon logs status messages and error messages to the system log or standard output, depending on the configuration settings. The daemon also provides options for adjusting the log verbosity and setting correction values for the PPS source.

Overall, ts2phc is a powerful and flexible tool for synchronizing the system clock with a PHC clock or an external PPS source, and it provides a wide range of advanced features for accurate timekeeping and diagnostics.

In general, an event listener is a component of a program that waits for and responds to specific events or signals. In the case of ts2phc, the daemon sets up event listeners to receive time stamp and pulse signals from the PTP and PPS sources, respectively.

When an event listener receives a signal, it triggers a corresponding action in the program. For example, when ts2phc receives a time stamp from the PTP source, it uses the time stamp to adjust the system clock. Similarly, when ts2phc receives a pulse signal from the PPS source, it uses the pulse signal to adjust the system clock.
The event listeners in ts2phc are implemented using various techniques, such as interrupt-driven I/O, polling, or signal handling. Interrupt-driven I/O involves setting up hardware interrupts to trigger the event listener when a signal is received. Polling involves periodically checking for the presence of a signal in a specific location. Signal handling involves registering a handler function to be called when a specific signal is received.
Overall, event listeners are an essential component of ts2phc and other programs that rely on precise timing and synchronization. By efficiently and accurately responding to time stamp and pulse signals from the PTP and PPS sources, ts2phc is able to maintain sub-microsecond accuracy in clock synchronization and ensure reliable timekeeping in a wide range of applications.
Specifically, If you use the "nmea" keyword to specify an external 1-PPS source that provides Time of Day (ToD) information via the RMC NMEA sentence, ts2phc will be able to synchronize the system clock based on the received ToD information, the process of ts2phc would work:

- Configuration: You would need to configure ts2phc to use the external 1-PPS source with the "nmea" keyword. You would also need to specify the device or interface that provides the RMC NMEA sentence, as well as other options such as the correction value for the PPS source.
- Initialization: Once ts2phc is configured, it initializes the 1-PPS source by opening a connection to the device or interface specified in the configuration file. The daemon sets up the 1-PPS source to generate a pulse signal at regular intervals, typically once per second.
- Signal detection: During operation, ts2phc continuously monitors the 1-PPS source for pulse signals. When a pulse signal is detected, the daemon records the time stamp of the pulse signal and uses it to adjust the system clock based on the ToD information provided by the RMC NMEA sentence.
- Correction: To ensure accuracy, ts2phc applies various correction values, such as delay and offset corrections, to compensate for any time drift or latency in the 1-PPS source. These correction values are typically specified in the configuration file.
- Synchronization: By continuously monitoring and adjusting the system clock based on the ToD information provided by the RMC NMEA sentence, ts2phc is able to synchronize the system time to the 1-PPS source with high accuracy.
- Logging and diagnostics: Throughout the synchronization process, ts2phc logs status messages and error messages to the system log or standard output,

depending on the configuration settings. The daemon also provides options for adjusting the log verbosity and setting correction values for the 1-PPS source, which can be useful for troubleshooting any issues that may arise.

6.1. Options

6.1.1. -a

The "-a" option is used with the ts2phc utility program to adjust the direction of synchronization automatically. Normally, when using ts2phc to synchronize PHC devices, the user must specify which device should be the reference clock and which devices should be the destination clocks. However, with the "-a" option, ts2phc can automatically determine which PHC device should be used as the reference clock and which devices should be used as destination clocks.

To do this, ts2phc queries the port states from the running instance of ptp4l, which is a program used to synchronize clocks on a network using the Precision Time Protocol (PTP). Based on the information obtained from ptp4l, ts2phc can determine which PHC device should be the reference clock and which devices should be the destination clocks.

A PPS source is a device that generates a Pulse-Per-Second (PPS) signal, which is a precise timing signal that is used to synchronize clocks on a system. In the context of PHC devices, a PPS source is typically a device that generates a PPS signal based on a highly accurate clock, such as a GPS receiver.

A PPS sink, on the other hand, is a device that receives a PPS signal and uses it to synchronize its clock. In the context of PHC devices, a PPS sink is typically a device that has a PHC clock that needs to be synchronized to a more accurate clock source.

In a typical setup, a PPS source (such as a GPS receiver) is connected to a PPS sink (such as a PHC device) via a physical connection, such as a serial cable. The PPS signal is transmitted over this connection and is used to synchronize the clocks on the PPS sink device.

It's worth noting that not all PHC devices have PPS capabilities, and not all PPS sources and sinks are PHC devices. However, PPS signals are commonly used in systems that require high-precision timing, such as in scientific experiments, financial trading, and telecommunications.

6.1.2. -c

The "-c" option is used with the ts2phc utility program to specify the PHC device that should be synchronized. The device can be identified either by its character device file, such as "/dev/ptp0", or by its associated network interface, such as "eth0".

When using the "-c" option, the user can specify one or more PHC devices to be synchronized by ts2phc. The option can be given multiple times, once for each PHC device that needs to be synchronized. For example, to synchronize two PHC devices with the names "ptp0" and "ptp1", the following command can be used:

```
ts2phc -c /dev/ptp0 -c /dev/ptp1
```

6.1.3. -f

The "-f" option is used with the ts2phc utility program to specify a configuration file to be used. By default, ts2phc does not read any configuration file, and the program must be run with the necessary command-line options to specify the PHC devices and other settings. However, by using the "-f" option, the user can specify a configuration file that contains the necessary options and settings for ts2phc.

6.1.4. -l

The "-l" option is used with the ts2phc utility program to set the maximum syslog level of messages that should be printed or sent to the system logger. Syslog is a standard logging mechanism used on Unix-like systems to collect and record system messages from various programs.

The print-level can be specified as an integer value between 0 and 7, corresponding to the syslog levels defined in the syslog.h header file:

- 0: LOG_EMERG (system is unusable)
- 1: LOG_ALERT (action must be taken immediately)
- 2: LOG_CRIT (critical conditions)
- 3: LOG_ERR (error conditions)
- 4: LOG_WARNING (warning conditions)
- 5: LOG_NOTICE (normal but significant condition)
- 6: LOG_INFO (informational messages)
- 7: LOG_DEBUG (debug-level messages)

The default print-level is 6, which corresponds to LOG_INFO. This means that informational messages and above (i.e., LOG_INFO, LOG_NOTICE, LOG_WARNING, LOG_ERR, LOG_CRIT, LOG_ALERT, and LOG_EMERG) will be printed or sent to the system logger.

To specify a different print-level, the "-l" option can be used followed by the desired integer value. For example, to set the print-level to LOG_DEBUG (level 7), the following command can be used: ts2phc -l 7. This will cause all messages, including debug-level messages, to be printed or sent to the system logger.

6.1.5. -m

The "-m" option is used with the ts2phc utility program to print log messages to the standard output. By default, ts2phc sends log messages to the system logger using the syslog mechanism, but with the "-m" option, log messages are printed to the standard output instead.
This can be useful for debugging or troubleshooting purposes, as it allows the user to see the log messages in real-time as they are generated by ts2phc. However, it's important to note that printing log messages to the standard output can generate a lot of output, especially if the program is running for an extended period of time or if there are a large number of log messages generated.

6.1.6. -q

The "-q" option is used with the ts2phc utility program to prevent log messages from being sent to the system logger. By default, ts2phc sends log messages to the system logger using the syslog mechanism, but with the "-q" option, log messages are suppressed and not sent to the system logger.
This can be useful in situations where the system logger is not needed or where the user wants to minimize the amount of logging generated by ts2phc. However, it's important to note that suppressing log messages can make it more difficult to troubleshoot issues or diagnose problems that may arise during the synchronization process.

6.1.7. -s

The "-s" option is used with the ts2phc utility program to specify the source of the Time of Day (ToD) data. ToD data is used to synchronize the clock of the PHC device to the external time source. If an external 1-PPS signal is used as the time source, the keyword "generic" can be used. If an external GPS device is used as the time source and provides ToD information via the RMC NMEA sentence, the keyword "nmea" can be used.

- To specify an external 1-PPS signal as the time source without ToD information, use the keyword "generic" with the "-s" option.
- To specify an external GPS device as the time source with ToD information provided through NMEA sentences, use the keyword "nmea" with the "-s" option.

6.1.8. -v

Prints the software version and exits.

6.2. Configuration file

The configuration file used by ts2phc is divided into sections, with each section containing settings for a specific aspect of the program. Each section is enclosed in brackets, and the name of the section is specified within the brackets.

Within each section, each setting is specified on a separate line, with the name of the option and the value separated by whitespace characters. Empty lines and lines starting with the "#" character are ignored and treated as comments.

Here's an example of a configuration file with two sections, one for the PHC devices and another for the time source:

```
[PHC]
# Specify the PHC devices to be synchronized
-c /dev/ptp0
-c /dev/ptp1

[Time Source]
# Specify the external GPS device as the time source
-s nmea
```

Note that the use of a configuration file can help simplify the process of running ts2phc with the desired options and settings, especially when multiple PHC devices and time sources are involved.

The configuration file used by ts2phc can contain two different types of sections:

- The global section, indicated by the name "[global]", which sets the program options and default time sink options. The options specified in this section apply to all PHC devices unless they are overridden by clock-specific sections.
- Clock-specific sections, which are named after the clock they configure (e.g., "[eth0]", "[ptp0]") and override the default options set in the global section. Each clock-specific section corresponds to a PHC device that is being synchronized.

In addition to these two types of sections, there can also be time source sections, which specify the source of the Time of Day (ToD) data. These sections are named after the time source they configure (e.g., "[GPS]") and specify options related to the external time source.

When ts2phc is run with a configuration file, any time sinks (destination) specified in the file do not need to be specified with the "-c" command-line option. Instead, the program reads the time sinks from the configuration file, and they are automatically used for synchronization.

6.3. Global Options

6.3.1. first_step_threshold

The "first_step_threshold" option in ts2phc specifies the maximum offset, in seconds, that the servo will correct by changing the clock frequency (or phase when using the nullf servo) instead of stepping the clock on the first update.

When ts2phc starts, it performs an initial synchronization between the time source and the PHC device, which is known as the first update. During this update, the "first_step_threshold" option specifies the maximum offset that the servo will correct without stepping the clock.

If the offset between the time source and the PHC device is greater than the "first_step_threshold" value, the servo will step the clock to correct the offset. If the offset is less than or equal to the "first_step_threshold" value, the servo will adjust the clock frequency (or phase) to correct the offset.

The default value for "first_step_threshold" is 0.00002 seconds (20 microseconds), which is a very small value that allows for precise synchronization without stepping the clock on start. However, depending on the specific use case, it may be necessary to adjust this value to achieve the desired level of accuracy.

It's also worth noting that if the "first_step_threshold" option is set to 0.0, the servo will not step the clock on start and will instead rely solely on frequency adjustments to correct any initial offset. This may be useful in some situations where stepping the clock on start could cause issues with other systems or devices that rely on the clock signal.

6.3.2. free_running

The "free_running" option in ts2phc specifies whether or not the PHC devices should be adjusted during the synchronization process. When "free_running" is set to 1, no PHC device will be adjusted, and they will continue to run at their current frequency. This can be useful in test scenarios, where you want to measure the synchronization accuracy of a group of local clocks without actually adjusting the clocks themselves.

When "free_running" is set to 0 (the default), the PHC devices will be adjusted during the synchronization process to reduce the offset between the local clock and the external time source. This is the normal behavior of ts2phc and is what allows it to provide accurate synchronization of local clocks.

It's worth noting that when "free_running" is set to 1, ts2phc will continue to output timing information, such as the offset between the local clock and the external time source, but it will not actually adjust the clock frequency or perform any other synchronization actions.

Overall, the "free_running" option can be a useful tool for testing and measuring the accuracy of local clocks, but it should be used with caution as it can result in unsynchronized clock signals if not used properly.
In the context of ts2phc, a test scenario refers to a situation where you want to evaluate or measure the behavior of the synchronization process without actually adjusting the clocks themselves. Test scenarios can be useful for a variety of purposes, such as:

- Measuring the accuracy of local clocks: By running ts2phc in a test scenario with the "free_running" option set to 1, you can measure the accuracy of local clocks without actually adjusting the clocks themselves. This can be useful for evaluating the performance of different clock sources or comparing the accuracy of different clock signals.
- Evaluating the behavior of the synchronization process: By running ts2phc in a test scenario with different configuration options, you can evaluate the behavior of the synchronization process under different conditions. For example, you might want to test how well ts2phc performs when synchronizing multiple PHC devices or when using different time sources.
- Debugging synchronization issues: By running ts2phc in a test scenario with debug options enabled, you can gather more detailed information about the synchronization process and identify any issues or errors that may be occurring.

To set up a test scenario with ts2phc, you would typically create a configuration file with the desired options and settings, including the "free_running" option set to 1 if you want to evaluate the behavior of the synchronization process without actually adjusting the clocks. You would then run ts2phc with the configuration file and monitor the output to gather data and evaluate the performance of the synchronization process.
It's worth noting that test scenarios can be complex and require careful planning and execution to ensure accurate and meaningful results. It's important to thoroughly understand the behavior of ts2phc and the options available, as well as the limitations of the hardware and software being used, in order to set up and run effective test scenarios.

6.3.3. leapfile

The "leapfile" option in ts2phc specifies the path to the current leap seconds definition file. This file contains information about the leap seconds that have been added or removed from Coordinated Universal Time (UTC) since 1972. Leap seconds are added periodically to keep UTC in sync with the rotation of the Earth.

In a Debian system, the leap seconds definition file is typically provided by the tzdata package and can be found at /usr/share/zoneinfo/leap-seconds.list. However, the path may be different on other systems or if the file has been installed in a custom location.
If the "leapfile" option is configured in ts2phc, the program will use the leap seconds definition file at the specified path, and it will reload the file if it is modified. This ensures that ts2phc always has the most up-to-date information about leap seconds and can accurately adjust the PHC devices to account for them.
If the "leapfile" option is not configured (the default), ts2phc will use a hard-coded table of leap seconds that reflects the known leap seconds on the date of the software's release. This may not be accurate if new leap seconds have been added since the software was released, so it is generally recommended to configure the "leapfile" option to ensure accurate synchronization.

6.3.4. logging_level

The "logging_level" option in ts2phc specifies the maximum logging level of messages that should be printed to the console or written to the log file.
There are several different logging levels available in ts2phc, ranging from the most severe (LOG_EMERG) to the most verbose (LOG_DEBUG). The available logging levels, in increasing order of severity, are:

- LOG_DEBUG
- LOG_INFO
- LOG_NOTICE
- LOG_WARNING
- LOG_ERR
- LOG_CRIT
- LOG_ALERT
- LOG_EMERG

When the "logging_level" option is set in ts2phc, messages with a severity level equal to or higher than the specified level will be printed or logged. For example, if the logging level is set to LOG_WARNING, messages with severity levels of LOG_WARNING, LOG_ERR, LOG_CRIT, LOG_ALERT, and LOG_EMERG will be printed or logged, while messages with severity levels of LOG_DEBUG, LOG_INFO, and LOG_NOTICE will be ignored.
The default logging level in ts2phc is LOG_INFO, which means that messages with severity levels of LOG_INFO, LOG_NOTICE, LOG_WARNING, LOG_ERR, LOG_CRIT, LOG_ALERT, and LOG_EMERG will be printed or logged. This provides a reasonable balance between verbosity and information content.
However, in some cases, it may be useful to adjust the logging level to either increase or decrease the amount of information that is printed or logged. For example, if you are debugging a synchronization issue, you may want to set the logging level to

LOG_DEBUG to enable more detailed logging of the synchronization process. Conversely, if you are running ts2phc in a production environment, you may want to set the logging level to LOG_WARNING or higher to reduce the amount of extraneous information that is printed or logged.

6.3.5. max_freqency

The "max_frequency" option in ts2phc specifies the maximum allowed frequency adjustment of the clock in parts per billion (ppb). This option is an additional limit to the maximum allowed by the hardware, and it can be used to prevent excessive frequency adjustments that could cause instability or other issues in the system.
The default value for "max_frequency" in ts2phc is 900000000 (90%), which means that the maximum allowed frequency adjustment is 90% of the hardware limit. If "max_frequency" is set to 0, the hardware limit will be used as the maximum allowed frequency adjustment.
For example, if the hardware limit for frequency adjustment is +/- 1000 ppb, and "max_frequency" is set to 900000000, ts2phc will limit the frequency adjustment to +/- 900 ppb, which is 90% of the hardware limit. If "max_frequency" is set to 0, ts2phc will use the full hardware limit of +/- 1000 ppb.
It's worth noting that the "max_frequency" option should be used with caution, as setting it too low could prevent ts2phc from making necessary frequency adjustments to maintain accurate synchronization. However, in some cases, it may be useful to set a lower limit to prevent excessive frequency adjustments that could cause issues in the system. The appropriate value for "max_frequency" will depend on the specific use case and the hardware being used.

6.3.6. message_tag

The "message_tag" option in ts2phc specifies the tag that is added to all messages printed to the standard output or system log. The tag provides additional information about the source of the message and can be useful for filtering and organizing log messages.
By default, the "message_tag" option is set to an empty string, which means that no tag is added to log messages. However, you can specify a tag by using the following command-line option: --message-tag=<tag>
Where "<tag>" is the tag you want to use for log messages. For example, you might use a tag like "ts2phc" to identify log messages generated by ts2phc.
When a tag is specified using the "message_tag" option, it will be added to all log messages printed to the standard output or system log. For example, a log message might look like this: ts2phc: adjusting clock frequency by 500 ppb
In this example, "ts2phc" is the tag that was specified using the "message_tag" option, and the message indicates that ts2phc is adjusting the clock frequency by 500 ppb.

6.3.7. step_threshold

The "step_threshold" option in ts2phc specifies the maximum offset, in seconds, that the servo will correct by changing the clock frequency instead of stepping the clock.
When ts2phc detects an offset between the local clock and the external time source, it uses a servo algorithm to adjust the clock frequency to reduce the offset. However, in some cases, the offset may be too large to be corrected by changing the clock frequency, and in these cases, ts2phc may need to step the clock to bring it into alignment with the external time source.
The "step_threshold" option allows you to control the threshold at which ts2phc switches from adjusting the clock frequency to stepping the clock. If the offset between the local clock and the external time source exceeds the "step_threshold" value, ts2phc will step the clock instead of adjusting the frequency.
The default value for "step_threshold" in ts2phc is 0.0, which means that ts2phc will never step the clock except on startup. If "step_threshold" is set to a nonzero value, ts2phc will use the specified value as the threshold for switching from frequency adjustment to clock stepping.
For example, if "step_threshold" is set to 1.0, and ts2phc detects an offset between the local clock and the external time source that exceeds 1.0 seconds, it will step the clock instead of adjusting the frequency to correct the offset.
It's worth noting that setting the "step_threshold" option too low can result in frequent clock stepping, which can be disruptive and may cause issues in the system. Conversely, setting the threshold too high can result in inaccurate synchronization if the offset exceeds the threshold. The appropriate value for "step_threshold" will depend on the specific use case and the hardware being used.

6.3.8. ts2phc.nmea_remote_host, ts2phc.nmea_remote_port

The "ts2phc.nmea_remote_host" and "ts2phc.nmea_remote_port" options in ts2phc specify the remote host and port that provide Time of Day (ToD) information when using the "nmea" PPS signal source.
When using the "nmea" PPS signal source, ts2phc can receive ToD information from a remote NMEA server instead of a local serial port. To use this feature, you can specify the remote host and port using the "ts2phc.nmea_remote_host" and "ts2phc.nmea_remote_port" options, respectively.
If both options are specified, ts2phc will use the remote connection in preference to the configured serial port. This can be useful in situations where it is not feasible to connect to a local serial port, or where a more accurate ToD source is available on a remote server.
By default, both options are set to an empty string, which means that ToD information will be obtained from a local serial port if the "nmea" PPS signal source is selected. If

you want to use a remote NMEA server instead, you can specify the remote host and port using the following command-line options:

```
--nmea-remote-host=<hostname>
--nmea-remote-port=<port>
```

Where "<hostname>" is the hostname or IP address of the remote NMEA server, and "<port>" is the port on which the server is listening for NMEA connections.
Overall, the "ts2phc.nmea_remote_host" and "ts2phc.nmea_remote_port" options are useful configuration options in ts2phc that allow you to obtain ToD information from a remote NMEA server when using the "nmea" PPS signal source.

6.3.9. ts2phc.nmea_serialport and ts2phc.nmea_baudrate

The "ts2phc.nmea_serialport" and "ts2phc.nmea_baudrate" options in ts2phc specify the serial port and baudrate in bps for the character device that provides Time of Day (ToD) information when using the "nmea" PPS signal source.
When using the "nmea" PPS signal source, ts2phc can receive ToD information from a local serial port. To use this feature, you can specify the serial port and baudrate using the "ts2phc.nmea_serialport" and "ts2phc.nmea_baudrate" options, respectively.
By default, the "ts2phc.nmea_serialport" option is set to "/dev/ttyS0", which is the default serial port on many Linux systems. The "ts2phc.nmea_baudrate" option is set to 9600 bps by default, which is a common baud rate for NMEA devices.
If both the "ts2phc.nmea_remote_host" and "ts2phc.nmea_remote_port" options are specified, ts2phc will use the remote connection in preference to the configured serial port. This can be useful in situations where a more accurate ToD source is available on a remote server.
However, if you want to use a local serial port instead, you can specify the serial port and baud rate using the following command-line options:

```
--nmea-serialport=<device>
--nmea-baudrate=<baudrate>
```

rWhere "<device>" is the path to the serial port device file, and "<baudrate>" is the desired baud rate in bps.
Overall, the "ts2phc.nmea_serialport" and "ts2phc.nmea_baudrate" options are useful configuration options in ts2phc that allow you to obtain ToD information from a local serial port when using the "nmea" PPS signal source.

6.3.10. ts2phc.perout_phase

The "ts2phc.perout_phase" option in ts2phc configures the offset between the beginning of the second and the PPS source's rising edge when using the PHC kind of PPS source.

The "perout_phase" option is used to fine-tune the synchronization between the PPS source and the system clock. The option specifies the phase offset between the start of the second and the rising edge of the PPS signal in nanoseconds. The supported range is 0 to 999999999 nanoseconds.

By default, the "perout_phase" option is set to 0 nanoseconds, which means that the rising edge of the PPS signal is assumed to occur exactly at the beginning of each second. However, if you need to adjust the phase offset between the PPS signal and the system clock, you can specify a non-zero value for the "perout_phase" option.

If the "perout_phase" option is left unspecified, ts2phc will not transmit the phase to the kernel. Instead, PPS will be requested to start at an absolute time equal to the nearest 2nd full second since the start of the program. This should yield the same effect, but may not work with drivers that do not support starting periodic output at an absolute time.

Overall, the "perout_phase" option is a useful configuration option in ts2phc that can help to fine-tune the synchronization between the PPS source and the system clock when using the PHC kind of PPS source.

6.3.11. ts2phc.pulsewidth

The "ts2phc.pulsewidth" option in ts2phc specifies the pulse width of the external PPS signal in nanoseconds.

The "pulsewidth" option is used to configure the duration of the PPS signal that is generated by an external PPS source. This option is only applicable when using an external PPS source, and it is ignored when using the PHC kind of PPS source.

By default, the "pulsewidth" option is set to 500000000 nanoseconds, which is equivalent to a pulse width of 500 ms. This value is appropriate for many PPS sources, but you may need to adjust it if your PPS source uses a different pulse width.

The supported range for the "pulsewidth" option is 1000000 to 990000000 nanoseconds. If the "pulsewidth" option is set outside of this range, ts2phc will print a warning message and use the default value instead.

If the "extts_polarity" option is set to "both", the "pulsewidth" option is used to detect and discard the time stamp of the unwanted edge. In this case, the pulse width should be set to the duration of the desired edge. For example, if the PPS source generates a pulse on both the rising and falling edges, and you only want to use the rising edge, you would set the "pulsewidth" option to the duration of the rising edge.

Overall, the "pulsewidth" option is a useful configuration option in ts2phc that allows you to configure the pulse width of the external PPS signal and customize the behavior of the edge detection algorithm.

6.3.12. ts2phc.tod_source

The "ts2phc.tod_source" option in ts2phc specifies the source of Time of Day (ToD) data.
By default, the "tod_source" option is set to "generic", which means that ts2phc will assume that the external 1-PPS signal does not provide any ToD information.
If you are using a PHC as the time source, you can specify the clock by its character device (e.g., "/dev/ptp0") or its associated network interface (e.g., "eth0"). In this case, the PHC will provide ToD information as part of the PTP protocol.
Alternatively, if you are using an external 1-PPS signal from a GPS device that provides ToD information via the RMC NMEA sentence, you can specify the "nmea" option for the "tod_source" parameter. This will configure ts2phc to extract the ToD information from the NMEA sentence and use it to synchronize the system clock.
Overall, the "tod_source" option is a useful configuration option in ts2phc that allows you to specify the source of ToD information and customize the behavior of the time synchronization algorithm.

6.3.13. se_syslog

The "se_syslog" option in ts2phc controls whether or not messages are printed to the system log.
By default, the "se_syslog" option is set to 1, which means that messages will be printed to the system log if enabled. If you want to disable this feature, you can set the "se_syslog" option to 0.
When enabled, messages are printed to the system log using the syslog(3) function. The log messages typically include information about the status of the ts2phc daemon, including error messages and debugging information.
Overall, the "se_syslog" option is a useful configuration option in ts2phc that allows you to control the verbosity of the log messages and customize the behavior of the daemon.

6.3.14. verbose

The "verbose" option in ts2phc controls whether or not messages are printed to the standard output.
By default, the "verbose" option is set to 0, which means that messages will not be printed to the standard output. If you want to enable this feature, you can set the "verbose" option to 1.

When enabled, messages are printed to the standard output using the printf() function. The log messages typically include information about the status of the ts2phc daemon, including error messages and debugging information.
Overall, the "verbose" option is a useful configuration option in ts2phc that allows you to control the verbosity of the log messages and customize the behavior of the daemon.

6.4. Time Sink Options

6.4.1. ts2phc.channel

The "ts2phc.channel" option in ts2phc allows you to select a particular external time stamping or periodic output channel to be used.
Some PHC devices feature programmable pins and one or more time stamping channels. The "channel" option allows you to select a specific channel to be used for time stamping or periodic output.
When using a PHC device as the PPS source, the "channel" option selects the periodic output channel. This option is useful when you have multiple channels available and want to select a specific channel for output.
By default, the "channel" option is set to 0, which means that the first channel is selected. If you want to use a different channel, you can specify the desired channel number using the "ts2phc.channel" option.
Overall, the "channel" option is a useful configuration option in ts2phc that allows you to select a specific channel for time stamping or periodic output when using a PHC device.

6.4.2. ts2phc.extts_correction

The "ts2phc.extts_correction" option in ts2phc allows you to specify a correction value to be added to each PPS time stamp generated by an external PPS source.
By default, the "extts_correction" option is set to 0, which means that no correction value is added to the PPS time stamps. If you need to apply a correction to the PPS time stamps, you can specify the desired correction value in nanoseconds using the "ts2phc.extts_correction" option.
The correction value can be positive or negative, depending on the desired adjustment. For example, if the external PPS source is consistently generating a signal that is slightly ahead of the system clock, you could apply a negative correction value to adjust the time stamps accordingly.
Overall, the "extts_correction" option is a useful configuration option in ts2phc that allows you to apply a correction to the PPS time stamps generated by an external PPS source.

6.4.3. ts2phc.extts_polarity

The "ts2phc.extts_polarity" option in ts2phc allows you to specify the polarity of the external PPS signal.

The "extts_polarity" option can be set to either "rising" or "falling" to indicate the polarity of the PPS signal. By default, the "extts_polarity" option is set to "rising", which means that ts2phc expects the PPS signal to have a rising edge.

Some PHC devices always time stamp both edges of the PPS signal. If you are using such a device and only want to use one of the edges, you can set the "extts_polarity" option to "both". In this case, be sure to set the "ts2phc.pulsewidth" option to the correct value to detect and ignore the unwanted edge.

Overall, the "extts_polarity" option is a useful configuration option in ts2phc that allows you to specify the polarity of the external PPS signal and customize the behavior of the edge detection algorithm.

6.4.4. ts2phc.master

The "ts2phc.master" option in ts2phc allows you to configure a PHC device as the source of the PPS signal.

By default, the "master" option is set to 0, which means that the PHC device is configured as a time sink, i.e., it synchronizes its clock to an external time source.

If you want to configure the PHC device as the source of the PPS signal, you can set the "master" option to 1. In this case, the PHC device will generate the PPS signal and provide it to other devices for synchronization.

Overall, the "master" option is a useful configuration option in ts2phc that allows you to configure a PHC device as the source of the PPS signal or as a time sink, depending on your requirements.

6.4.5. ts2phc.pin_index

The "ts2phc.pin_index" option in ts2phc allows you to specify the pin index to be used for external time stamping or periodic output on PHC devices that feature programmable pins.

By default, the "pin_index" option is set to 0, which means that the first pin is selected. If you want to use a different pin for external time stamping or periodic output, you can specify the desired pin index using the "ts2phc.pin_index" option.

The available pin indices depend on the specific PHC device that you are using. You should consult the documentation for your device to determine the available pin indices and their associated functions.

Overall, the "pin_index" option is a useful configuration option in ts2phc that allows you to select a specific pin for external time stamping or periodic output on PHC devices that feature programmable pins.

6.5. Ts2phc source code

Before research about the functions in ts2phc, we need deeply understand about the structures appeared in source code.

6.5.1. struct ts2phc_clock

```
struct ts2phc_clock {
LIST_ENTRY(ts2phc_clock) list;
clockid_t clkid;
int fd;
int phc_index;
enum port_state state;
enum port_state new_state;
struct servo *servo;
enum servo_state servo_state;
char *name;
bool no_adj;
bool is_target;
bool is_ts_available;
tmv_t last_ts;
};
```

The ts2phc_clock struct is used to represent a clock in the ts2phc program. It contains various fields that hold information about the clock, such as its clock ID, file descriptor, PHC index, current state, servo state, and name.

The LIST_ENTRY(ts2phc_clock) list field is used to link multiple ts2phc_clock structures together into a linked list. This allows the program to easily iterate over all the clocks that it is managing.

The servo field points to a servo structure that holds information about the clock's servo, which is used for servo control and clock discipline. The servo_state field holds the current state of the servo.

The no_adj field is a boolean value that indicates whether the clock should be adjusted or not. The is_target field is also a boolean value that indicates whether the clock is being targeted for adjustment.

The is_ts_available field is a boolean value that indicates whether the clock has a valid timestamp available. The last_ts field holds the timestamp of the last PPS signal received by the clock, which is used for timestamping purposes.

Overall, the ts2phc_clock struct plays a central role in the ts2phc program, as it holds information about the clocks that the program is managing and is used extensively throughout the program's codebase.

6.5.2. struct ts2phc_port

```
struct ts2phc_port {
        LIST_ENTRY(ts2phc_port) list;
```

```
        unsigned int number;
        enum port_state state;
        struct ts2phc_clock *clock;
};
```

The ts2phc_port struct is used to represent a PTP port in the ts2phc program. It contains various fields that hold information about the port, such as its port number, current state, and a pointer to the ts2phc_clock structure that the port is associated with.

The LIST_ENTRY(ts2phc_port) list field is used to link multiple ts2phc_port structures together into a linked list. This allows the program to easily iterate over all the ports that it is managing.

The number field holds the port number of the PTP port. The state field holds the current state of the port.

The clock field is a pointer to the ts2phc_clock structure that the port is associated with. This allows the program to associate a clock with a particular port and manage the clock's synchronization status through that port.

Overall, the ts2phc_port struct is an important data structure in the ts2phc program, as it allows the program to manage the synchronization status of multiple PTP ports and the clocks associated with them. The linked list of ports is used extensively throughout the program's codebase to keep track of all the ports that are being managed and to update their synchronization status as needed.

6.5.3. struct ts2phc_private

```
struct ts2phc_private {
        struct ts2phc_pps_source *src;
        STAILQ_HEAD(sink_ifaces_head, ts2phc_pps_sink) sinks;
        unsigned int n_sinks;
        struct ts2phc_sink_array *polling_array;
        tmv_t perout_phase;
        struct config *cfg;
        struct pmc_agent *agent;
        struct ts2phc_clock *ref_clock;
        bool state_changed;
        LIST_HEAD(port_head, ts2phc_port) ports;
        LIST_HEAD(clock_head, ts2phc_clock) clocks;
};
```

The ts2phc_private struct is used to hold program-specific data for the ts2phc program. It contains various fields that hold information about the program's configuration, PPS source and sinks, clocks, ports, and synchronization state.

The src field holds a pointer to the ts2phc_pps_source structure that represents the PPS source for the program. This source is used to collect PPS data and synchronize the system clocks.
The sinks field is a linked list of ts2phc_pps_sink structures that represent the PPS sinks for the program. These sinks are used to output the synchronized PPS signal to external devices or applications.
The n_sinks field holds the number of PPS sinks that the program is currently using.
The polling_array field is a pointer to a ts2phc_sink_array structure that is used to manage the polling of the PPS sinks.
The perout_phase field holds the current phase offset of the PPS output signal.
The cfg field holds a pointer to the configuration data for the program, which is loaded from a configuration file.
The agent field holds a pointer to a PMC agent, which is used for clock control and monitoring.
The ref_clock field is a pointer to the reference clock that the program is using for synchronization. This clock is typically the most accurate and stable clock in the system.
The state_changed field is a boolean value that indicates whether the program's synchronization state has changed since the last iteration of the program loop.
The ports field is a linked list of ts2phc_port structures that represent the PTP ports that the program is managing. These ports are used for clock synchronization and management.
The clocks field is a linked list of ts2phc_clock structures that represent the clocks that the program is managing. These clocks are used for clock discipline and synchronization.
Overall, the ts2phc_private struct is a central data structure in the ts2phc program, as it holds information about the program's configuration, PPS sources and sinks, clocks, ports, and synchronization state. The various fields of this struct are used extensively throughout the program's codebase to manage the program's behavior and state.

6.5.4. ts2phc_pps_source struct

```
struct ts2phc_pps_source {
        void (*destroy)(struct ts2phc_pps_source *src);
        int (*getppstime)(struct ts2phc_pps_source *src, struct timespec *ts);
        struct ts2phc_clock *(*get_clock)(struct ts2phc_pps_source *src);
};
```

The ts2phc_pps_source struct is an abstract base struct used to represent a PPS source in the ts2phc program. It defines three function pointers that must be implemented by any concrete PPS source struct that is used in the program.

The destroy function pointer is used to clean up any resources that were allocated by the PPS source struct during its lifetime. This function is typically called when the program is shutting down and needs to free any resources that were allocated by the PPS source struct.
The getppstime function pointer is used to retrieve the current time from the PPS source as a struct timespec value. This function is typically called periodically by the program to obtain updated PPS data and synchronize the system clocks with that data.
The get_clock function pointer is used to retrieve a pointer to the ts2phc_clock structure that is associated with the PPS source. This allows the program to manage the synchronization status of the system clocks based on the PPS data that is collected from the source.
Overall, the ts2phc_pps_source struct is an important part of the ts2phc program's architecture, as it defines the interface that must be implemented by all concrete PPS source structs used in the program. This allows the program to be easily extended to support different types of PPS sources in the future, while maintaining a consistent interface for managing those sources.

6.5.5. ts2phc_nmea_pps_source struct

```
struct ts2phc_nmea_pps_source {
        struct ts2phc_pps_source pps_source;
        struct config *config;
        struct lstab *lstab;
        pthread_t worker;
        /* Protects anonymous struct fields, below, from concurrent access. */
        pthread_mutex_t mutex;
        struct {
                struct timespec local_monotime;
                struct timespec local_utctime;
                struct timespec rmc_utctime;
                bool rmc_fix_valid;
        };
};
```

The ts2phc_nmea_pps_source struct is used to represent a NMEA PPS source in the ts2phc program. It contains various fields that hold information about the NMEA PPS source, such as configuration data, a leap second table, and synchronization state.
The pps_source field is a structure that holds function pointers for the getppstime and destroy functions, which are used to interact with the NMEA PPS source.
The config field holds a pointer to the configuration data for the program, which is loaded from a configuration file.

The lstab field holds a pointer to a leap second table (lstab) generated from the configuration file.
The worker field holds a pthread thread ID for the worker thread that is used to monitor the NMEA PPS source status.
The mutex field is a pthread mutex that is used to protect the anonymous struct fields below it from concurrent access.
The anonymous struct fields below the mutex field hold various synchronization state information, including the local monotime, local UTC time, the RMC (recommended minimum) sentence UTC time, and whether the RMC fix is valid.
Overall, the ts2phc_nmea_pps_source struct is an important data structure in the ts2phc program, as it holds information about the NMEA PPS source and is used extensively throughout the program's codebase to collect PPS data and synchronize the system clocks based on that data.

6.5.6. lstab struct

```
struct lstab {
        struct epoch_marker lstab[N_LEAPS];
        uint64_t expiration_utc;
        const char *leapfile;
        time_t lsfile_mtime;
        int length;
};
struct epoch_marker {
        int offset;      /* TAI - UTC offset of epoch */
        uint64_t ntp;  /* NTP time of epoch */
        uint64_t tai;   /* TAI time of epoch */
        uint64_t utc;  /* UTC time of epoch */
};
```

The lstab struct is used to represent a leap second table in the ts2phc program. It contains various fields that hold information about the leap second table, including the epoch markers, expiration time, file name, modification time, and number of entries.
The lstab field is an array of epoch_marker structures that represent the epoch markers for the leap second table. Each epoch_marker structure contains information about a leap second, including the UTC time when the leap second occurs, and the offset between TAI and UTC at that time.
The expiration_utc field holds the UTC time when the leap second table will expire and need to be updated.
The leapfile field holds the name of the file that contains the leap second table.
The lsfile_mtime field holds the modification time of the leap second table file.

The length field holds the number of entries in the leap second table.
Overall, the lstab struct is an important data structure in the ts2phc program, as it holds information about the leap second table and is used extensively throughout the program's codebase to synchronize the system clocks based on that table. The leap second table is periodically checked for updates, and the program uses the information in the table to adjust the system clocks accordingly.

6.5.7. struct nmea_parser

```
enum nmea_state {
        NMEA_IDLE,
        NMEA_HAVE_DOLLAR,
        NMEA_HAVE_STARTG,
        NMEA_HAVE_STARTX,
        NMEA_HAVE_BODY,
        NMEA_HAVE_CSUMA,
        NMEA_HAVE_CSUM_MSB,
        NMEA_HAVE_CSUM_LSB,
        NMEA_HAVE_PENULTIMATE,
};
struct nmea_parser {
        char sentence[NMEA_MAX_LENGTH + 1];
        char payload_checksum[3];
        enum nmea_state state;
        uint8_t checksum;
        int offset;
};
```

The nmea_parser struct is used to represent a NMEA sentence parser in the ts2phc program. It contains various fields that hold information about the parser state, including the current sentence, the payload checksum, the parser state, the calculated checksum, and the offset within the sentence.
The sentence field is a character array that holds the current NMEA sentence being parsed.
The payload_checksum field is a character array that holds the checksum for the sentence payload.
The state field is an enumeration that holds the current state of the parser, which can be one of several states, such as NMEA_STATE_WAITING_DOLLAR, NMEA_STATE_PARSING_PAYLOAD,...
The nmea_state enum defines the different states that a NMEA sentence parser in the ts2phc program can be in while parsing a NMEA sentence.

- NMEA_IDLE: The parser is in an idle state, waiting for the start of a new sentence.

- NMEA_HAVE_DOLLAR: The parser has detected the start of a new sentence, indicated by the '$' character.
- NMEA_HAVE_STARTG: The parser has detected the start of a new GGA (Global Positioning System Fix Data) sentence, indicated by the 'G' character.
- NMEA_HAVE_STARTX: The parser has detected the start of a new RMC (Recommended Minimum Specific GNSS Data) or ZDA (Time & Date) sentence, indicated by the 'R' or 'Z' character.
- NMEA_HAVE_BODY: The parser is currently parsing the body of the NMEA sentence.
- NMEA_HAVE_CSUMA: The parser has detected the start of the checksum field, indicated by the '*' character.
- NMEA_HAVE_CSUM_MSB: The parser is currently parsing the most significant byte of the checksum.
- NMEA_HAVE_CSUM_LSB: The parser is currently parsing the least significant byte of the checksum.
- NMEA_HAVE_PENULTIMATE: The parser has finished parsing the sentence and is waiting for the carriage return character to end the sentence.

The checksum field is an 8-bit integer that holds the calculated checksum for the sentence.
The offset field is an integer that holds the current offset within the sentence that is being parsed.
Overall, the nmea_parser struct is an important data structure in the ts2phc program, as it is used to parse NMEA sentences and extract timing information from them. The parser is used extensively throughout the program's codebase to collect timing data from NMEA sources and synchronize the system clocks based on that data.

6.5.7. nmea_rmc struct

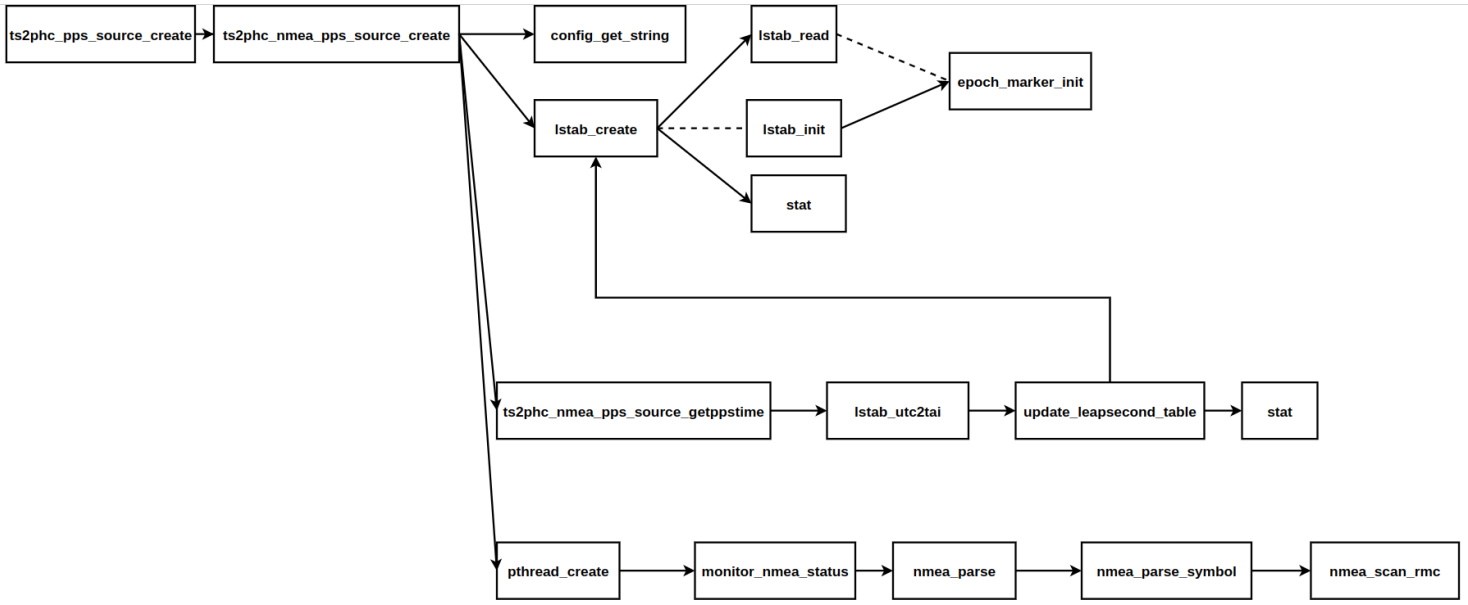
```
struct nmea_rmc {
        struct timespec ts;
        bool fix_valid;
};
```

The nmea_rmc struct is used to represent timing information extracted from a RMC (Recommended Minimum Specific GNSS Data) NMEA sentence in the ts2phc program. It contains two fields that hold information about the timing data extracted from the sentence.
The ts field is a struct timespec value that holds the UTC time extracted from the RMC sentence. This value is used to synchronize the system clocks with the UTC time obtained from the NMEA source.

The fix_valid field is a boolean value that indicates whether the fix obtained from the RMC sentence is valid or not. If the fix is not valid, the timing information obtained from the sentence may not be accurate or reliable.
Overall, the nmea_rmc struct is an important data structure in the ts2phc program, as it holds timing information extracted from a specific type of NMEA sentence (RMC), and is used to synchronize the system clocks based on that data. The ts field is particularly important, as it provides the UTC time that is used to synchronize the system clocks with the NMEA source.

6.6. Create Functions

Firstly, we will see the diagram which describes the flow of source code.



6.6.1. Main function

The main() function you provided is the entry point of the ts2phc daemon program. It is responsible for parsing the command line arguments, initializing the program, and running the main loop. Here is a brief summary of what the main() function does:

- Initialize variables and the configuration file: The function initializes several local variables, including the configuration file (cfg) and a private data structure (priv) that contains various program-specific data.
- Parse command line arguments: The function uses getopt_long() to parse the command line arguments. The available options are specified with config_long_options(), which retrieves the long options defined in the configuration file. The function then sets various program options based on the command line arguments, such as the logging level and whether to use syslog.
- Read the configuration file: If the -f option is specified, the function reads the configuration file specified by the option. The function then sets various

program options based on the configuration file, such as the PPS source and sinks.
- Initialize the program: The function initializes the program by setting up signal handlers and creating a PMC agent. It also initializes the PPS sinks and source, and opens a connection to the PPS source device or interface.
- Enter the main loop: The function enters the main loop, which runs until an exit condition is met. During each iteration of the loop, the program collects PPS signal data from the PPS sinks, collects timestamp data from the PPS source, and synchronizes the system clocks based on the collected data.
- Clean up and exit: Once the main loop exits, the function performs various cleanup tasks, such as closing open connections and freeing memory, and then exits the program with a return code indicating success or failure.

Overall, the main() function serves as the central control for the ts2phc daemon program, coordinating the various program components and ensuring that the program runs smoothly and accurately.

6.6.2. ts2phc_pps_source_create()

The ts2phc_pps_source_create() function takes three arguments: a pointer to a ts2phc_private structure that holds program-specific data, a string dev that specifies the device or interface to use as a PPS source, and an enumeration value type that specifies the type of PPS source to create.

The function creates a PPS source based on the specified type and returns a pointer to the resulting ts2phc_pps_source structure. The possible PPS source types are:

- TS2PHC_PPS_SOURCE_GENERIC: Creates a generic PPS source using the ts2phc_generic_pps_source_create() function.
- TS2PHC_PPS_SOURCE_NMEA: Creates a NMEA PPS source using the ts2phc_nmea_pps_source_create() function.
- TS2PHC_PPS_SOURCE_PHC: Creates a PHC PPS source using the ts2phc_phc_pps_source_create() function.

Which specific function is called depends on the value of the type argument.

If the PPS source creation fails, the function returns NULL.

Overall, this function plays a key role in setting up the PPS source for the program, and allows the program to select the appropriate PPS source type based on the configuration.

6.6.3. ts2phc_nmea_pps_source_create()

The ts2phc_nmea_pps_source_create() function takes two arguments: a pointer to a ts2phc_private structure that holds program-specific data, and a string dev that specifies the device or interface to use as a NMEA PPS source.

The function creates a NMEA PPS source and returns a pointer to the resulting ts2phc_pps_source structure. The function first allocates memory for a ts2phc_nmea_pps_source structure and initializes it with the necessary configuration data, including a leap second table (lstab) generated from a configuration file specified in the priv structure.

The function then sets the function pointers for the getppstime and destroy functions of the ts2phc_pps_source structure to the appropriate NMEA-specific implementations (ts2phc_nmea_pps_source_getppstime() and ts2phc_nmea_pps_source_destroy() respectively).

The function also initializes a mutex and creates a worker thread using the monitor_nmea_status() function to continuously monitor the NMEA PPS source status.

If any of the initialization steps fail, the function frees any allocated memory and returns NULL.

A worker thread is a separate thread of execution that is created by a program to perform a specific task in the background. In the context of the ts2phc program, the worker thread is created by the ts2phc_nmea_pps_source_create() function to continuously monitor the NMEA PPS source status.

When the worker thread is created, it starts running the monitor_nmea_status() function in the background. This function periodically reads data from the NMEA PPS source and updates the program's internal state accordingly. The worker thread runs independently of the main program loop and can run concurrently with other threads that may be performing other tasks.

By using a worker thread to monitor the NMEA PPS source, the program can ensure that PPS data is continuously collected and processed, even if the main program loop is blocked or delayed by other tasks.

The pthread_create() function is used to create the worker thread, which takes a function pointer (monitor_nmea_status() in this case) and the appropriate arguments as input. Once the thread is created, it runs in the background until it is either explicitly terminated or the program exits. In this case, the thread is terminated when the program exits or when an error occurs during thread creation.

It is important to manage worker threads carefully to avoid race conditions, deadlocks, or other synchronization issues. In the ts2phc program, the worker thread and the main program loop interact with shared data structures, so proper synchronization mechanisms such as mutexes are used to ensure that data is accessed safely and without conflicts.

6.6.4.lstab_create

The role of the lstab_create function is to create a new lstab object, which represents a leap second table. The function allocates memory for the lstab object using calloc, sets its initial values, and reads the leap second table from a file if a filename is provided.
By creating a new lstab object, the function allows the caller to work with a leap second table and perform operations such as adding, deleting, or modifying leap seconds. The leap second table is used by applications that need to convert between Coordinated Universal Time (UTC) and International Atomic Time (TAI), which are two different time scales that differ by an integral number of leap seconds.
The stat function is a system call that retrieves information about a file specified by its name and stores that information in a struct stat object pointed to by the second argument.
In the code you provided, stat(lstab->leapfile, &statbuf) retrieves the file status information for the file specified in the leapfile member of the lstab structure. The retrieved information is stored in the statbuf structure, which is passed as a second argument to the stat function.
The & operator is used to pass the address of the statbuf structure to the stat function, so that the function can modify the contents of the structure with the file status information.
After the stat function returns, the statbuf structure contains information such as the file's size, ownership, permissions, and timestamps. In the code you provided, the function retrieves the modification time of the file from the statbuf structure and stores it in the lstab->lsfile_mtime member of the lstab structure.
Overall, the stat function is used to obtain information about a file, which can be useful for various file-related operations.
The lstab_create function is an important part of a leap second table library that allows applications to handle leap seconds correctly and ensure accurate timekeeping.

6.6.5. epoch_marker_init &lstab_init

The epoch_marker_init function initializes an epoch_marker structure, which represents a specific point in time on three different time scales: NTP (Network Time Protocol) time, UTC (Coordinated Universal Time), and TAI (International Atomic Time). The function takes a uint64_t value val as input, which represents the number of NTP seconds since January 1, 1900, and an integer offset, which represents the number of leap seconds to apply for converting between UTC and TAI.
The function computes the corresponding values of UTC and TAI time by subtracting the NTP-UTC offset and the leap second offset, respectively, from the NTP time value. The function stores the resulting values in the ntp, utc, and tai members of the epoch_marker structure, and also stores the leap second offset in the offset member.

The lstab_init function initializes a lstab structure, which represents a leap second table. The function initializes the lstab structure by creating a set of epoch_marker structures, one for each historical leap second, and storing them in an array pointed to by the lstab member of the lstab structure. The epoch_marker_init function is called for each epoch_marker structure to initialize it with the correct NTP time, UTC time, and TAI time values.
The lstab_init function also sets the expiration time of the leap second table by subtracting the NTP-UTC offset from the expiration_date_ntp constant, and stores the result in the expiration_utc member of the lstab structure. Finally, the function sets the length member of the lstab structure to the number of historical leap seconds, which is equal to N_HISTORICAL_LEAPS.
In summary, epoch_marker_init initializes a single epoch_marker structure, while lstab_init initializes a lstab structure containing an array of epoch_marker structures. The epoch_marker structures represent specific points in time on different time scales, while the lstab structure represents a table of historical leap seconds used for converting between different time scales.

6.6.6. ts2phc_nmea_pps_source_getppstime

The ts2phc_nmea_pps_source_getppstime function is part of a PPS (Pulse Per Second) synchronization implementation that uses NMEA (National Marine Electronics Association) messages from a GPS receiver to synchronize the system clock to UTC time. The function is called by the PPS source module to get the current system time based on the NMEA messages received from the GPS receiver.
The function retrieves the current local time and the latest valid RMC message from the GPS receiver. It then computes the UTC time based on the timestamp in the RMC message and the time difference between the local time and the RMC message timestamp. The UTC time is converted to TAI time using a leap second table, which is stored in the lstab member of the ts2phc_nmea_pps_source structure.
The function returns the TAI time in the timespec structure pointed to by the ts input parameter. If there is no valid RMC message available or the RMC message is stale (older than MAX_RMC_AGE), the function returns an error code (-1). If there is an error in converting the UTC time to TAI time using the leap second table, the function returns an error code and prints an error message to the log.
Overall, the ts2phc_nmea_pps_source_getppstime function is an important part of a PPS synchronization implementation that allows accurate timekeeping by synchronizing the system clock to UTC time with the help of NMEA messages from a GPS receiver and a leap second table.

6.6.7. lstab_utc2tai

The lstab_utc2tai function is used to convert a UTC (Coordinated Universal Time) timestamp to TAI (International Atomic Time), taking into account any leap seconds that have occurred since January 1, 1972. The function takes a lstab structure pointer, which represents a leap second table, a uint64_t value utctime, which represents the number of seconds since January 1, 1970, and an int pointer tai_offset, which is used to return the number of leap seconds to add to the UTC time to obtain the corresponding TAI time.

The function first checks if the leap second table needs to be updated, and returns an error code if the update fails. It then searches the leap second table to find the epoch, i.e., the historical leap second entry, that corresponds to the UTC timestamp. The function searches the leap second table from the latest epoch to the earliest epoch, and returns an error code if no epoch is found.

If an epoch is found, the function determines the TAI offset for that epoch by retrieving the offset member of the corresponding epoch_marker structure. The function also checks if the UTC timestamp corresponds to a leap second boundary, i.e., if it is one second less than the UTC timestamp of the next epoch. If so, the function returns an error code indicating that the UTC timestamp is ambiguous.

Finally, the function checks if the UTC timestamp is after the expiration date of the leap second table, and returns an error code if so. Otherwise, the function computes the TAI offset by setting the tai_offset pointer to the TAI offset of the epoch and returns a success code.

Overall, the lstab_utc2tai function is an important part of a leap second table library that allows applications to convert between UTC and TAI time correctly and ensure accurate timekeeping.

6.6.8. update_leapsecond_table

This function updates a leap second table stored in a file specified by the leapfile field of the input lstab structure. The function first checks if the leapfile field is not null, and if it is null, it returns without doing anything. It then uses the stat function to get the status information of the file specified by leapfile, and if this operation fails, the function returns with an error code. If the file has not been modified since the last time the leap second table was updated (as determined by comparing the modification time of the file with the lsfile_mtime field of the input lstab structure), the function returns without doing anything. Otherwise, the function destroys the existing lstab structure, opens the leapfile file, creates a new lstab structure, and returns it. If any errors occur during this process, the function returns with an error code.

6.6.9. monitor_nmea_status

This function monitors the status of a remote NMEA source or a local NMEA serial port, parses the incoming NMEA messages, and updates the ts2phc_nmea_pps_source structure pointed to by the input argument arg with the parsed information. The function creates a new nmea_parser object and initializes a pollfd structure pfd with the file descriptor of the NMEA source or serial port to be monitored, as well as a timeout value tmo. It then reads from the NMEA source or serial port using poll and read system calls, and parses the received NMEA messages using the nmea_parse function provided by the nmea_parser object. If the parsing is successful, the function updates the ts2phc_nmea_pps_source structure pointed to by arg with the parsed information, including the local monotime, local UTC time, RMC UTC time, and RMC fix validity. The function also uses the adjtimex system call to adjust the system clock using the parsed time information.

The function runs in a loop until the is_running function returns false, which indicates that the program is being shut down. If any errors occur during the monitoring or parsing process, the function logs an error message and continues to the next iteration of the loop. When the loop is exited, the function destroys the nmea_parser object and closes the NMEA source or serial port.

6.6.10. nmea_parse

This function attempts to parse an NMEA message of the RMC (Recommended Minimum Specific GNSS Data) type using the stateful nmea_parser object pointed to by np, given a pointer to the message in a character buffer ptr of length buflen. The function uses nmea_parse_symbol and nmea_scan_rmc functions provided by the nmea_parser object to parse the message. If the message is successfully parsed, the function fills the struct nmea_rmc pointed to by result with the parsed data and sets the integer pointed to by parsed to the number of characters parsed from the input buffer. If the message cannot be parsed, the function sets parsed to the total number of characters in the input buffer and returns with an error code -1. The function operates in a loop, parsing symbols one by one until the end of the buffer is reached or the message is successfully parsed.

6.6.11. nmea_parse_symbol

This function is a helper function used by the nmea_parse function to parse individual symbols of an NMEA message. The function updates the state of the stateful nmea_parser object pointed to by np based on the symbol c passed as an argument. The nmea_parser object is used to keep track of the parsing state and accumulate the message payload.

The function operates in a switch statement that handles the different possible states of the nmea_parser object. When the function is initially called, the nmea_parser object

is in the NMEA_IDLE state and waits for a dollar sign ('$') to start a new message. When a dollar sign is received, the object transitions to the NMEA_HAVE_DOLLAR state. If the next symbol is 'G', the object transitions to the NMEA_HAVE_STARTG state and accumulates the received characters into the message buffer. If any of the symbols received after the dollar sign are not 'G', the function resets the state of the nmea_parser object and returns an error.
When the nmea_parser object is in the NMEA_HAVE_STARTG state, it waits for the next symbol to be 'P', indicating the start of a specific type of NMEA message. If the next symbol is 'P', the object transitions to the NMEA_HAVE_STARTX state and accumulates the received characters into the message buffer. When the nmea_parser object is in the NMEA_HAVE_STARTX state, it continues to accumulate characters into the message buffer until it encounters a '*' character, which indicates the end of the message payload.
When the nmea_parser object is in the NMEA_HAVE_BODY state, it accumulates the message payload until it encounters a '*' character, indicating the start of the message checksum. The object transitions to the NMEA_HAVE_CSUMA state and stores the first byte of the checksum. It then transitions to the NMEA_HAVE_CSUM_MSB state and stores the second byte of the checksum. When the nmea_parser object is in the NMEA_HAVE_CSUM_LSB state, it expects to encounter a carriage return ('\r') character followed by a line feed ('\n') character, indicating the end of the message. If the received symbol is a line feed, the function returns success (0) to indicate that the message was successfully parsed. If the received symbol is not a line feed, the function resets the state of the nmea_parser object and returns an error.
The nmea_reset and nmea_accumulate functions are helper functions used by this function to reset the state of the nmea_parser object and accumulate characters into the message buffer, respectively.

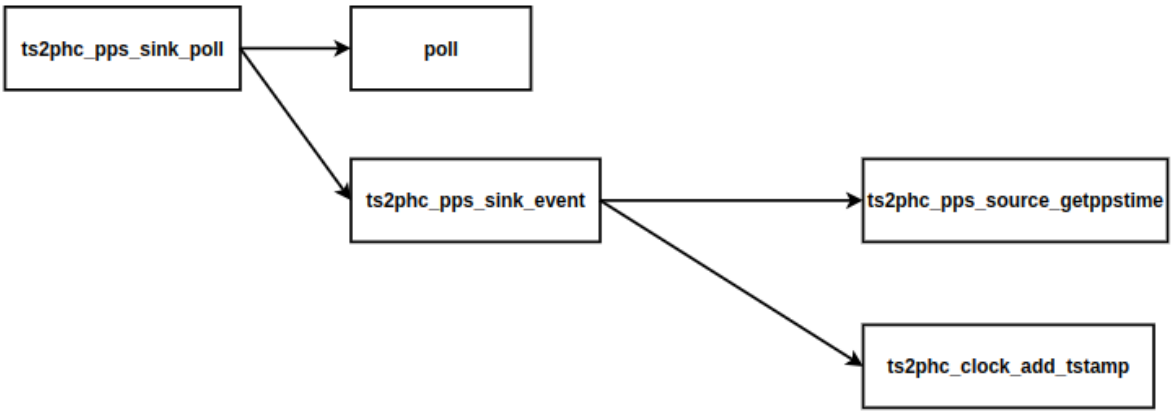
6.6.12. nmea_scan_rmc

This function parses the payload of an NMEA RMC (Recommended Minimum Specific GNSS Data) message that has been accumulated in the nmea_parser object pointed to by np. The function fills a struct nmea_rmc pointed to by result with the parsed data, including the UTC time stamp, fix validity, and time in nanoseconds. The function returns 0 if the parsing is successful and -1 otherwise.
The function first extracts the checksum from the np->payload_checksum buffer and compares it with the calculated checksum stored in the np->checksum field of the nmea_parser object. If the checksums do not match, the function returns an error.
The function then uses sscanf to parse the time and fix status from the message payload. It first attempts to parse the time in the format HHMMSS.sss, where HH is the hour, MM is the minute, SS is the second, and sss is the millisecond. If this format

is not found, the function attempts to parse the time in the format HHMMSS. If either of these formats is not found, the function returns an error. The function then uses sscanf again to parse the date in the format DDMMYY.
Next, the function converts the parsed time and date information into a Unix timestamp, which is stored in the result->ts field as a struct timespec object. The function also stores the parsed millisecond value in the result->ts.tv_nsec field as nanoseconds.
Finally, the function sets the result->fix_valid flag to true if the fix status is 'A' (valid fix) and false otherwise.
Overall, this function is responsible for parsing and extracting important information from an NMEA RMC message and storing it in a structured format for further processing by the calling program.

6.7. Polling

After ts2phc completed creating the pps source, it will poll the events. The process is described as below image:



6.7.1 ts2phc_pps_sink_poll

This function polls a set of PPS (Pulse Per Second) sinks for events and collects any events that occur within a timeout period. The function is called repeatedly by the ts2phc_pps_thread function to ensure that all sinks have reported an event before proceeding to the next cycle of the loop.
The function takes a struct ts2phc_private pointer as an argument, which contains information about the polling state and the set of sinks to be polled. The function uses the poll system call to wait for events on all the sinks in the polling_array until either an event occurs on all sinks or a timeout of two seconds is reached.
If the poll call returns an error, the function returns an error code. If the poll call returns zero, indicating that no event occurred within the timeout, the function returns 0.

If an event occurs on a sink, the function calls the ts2phc_pps_sink_event function with the sink and the private data pointer as arguments to handle the event. The ts2phc_pps_sink_event function updates the sink state and returns an enum value indicating whether the event should be ignored, handled, or if there was an error. The function collects the events that occurred on each sink, regardless of whether they will be ignored later.
The function proceeds to the next cycle of the loop only after all sinks have reported an event. If any sink reported an event that should be ignored, the function returns 0, indicating that no further processing is needed. If all sinks reported valid events, the function returns 1 to indicate that the next cycle of the loop should be executed.
Overall, this function is responsible for polling a set of PPS sinks for events and collecting the events that occur within a timeout period, ensuring that all sinks have reported an event before proceeding to the next cycle of the loop.

6.7.2 ts2phc_pps_sink_event

This function handles an external time stamp (EXTTS) event on a PPS sink by reading the event data from the sink's file descriptor, checking the event's channel, and calculating the corresponding time stamp. The function is called by the ts2phc_pps_sink_poll function for each sink that reports an event.
The function takes a struct ts2phc_private pointer and a struct ts2phc_pps_sink pointer as arguments. The struct ts2phc_pps_sink structure represents a PPS sink, which contains information about the sink's file descriptor, channel number, polarity, and correction value. The function also uses the struct ptp_extts_event structure to read the event data from the sink's file descriptor.
The function first reads the event data from the sink's file descriptor using the read system call. If the read operation fails to read the expected number of bytes, the function returns an error code. The function then checks if the event occurred on the expected channel by comparing the event.index field with the sink->pin_desc.chan field. If the channels do not match, the function returns an error code.
The function then attempts to retrieve the corresponding system time stamp from the PPS source using the ts2phc_pps_source_getppstime function. If the time stamp is not valid, the function returns 0, indicating that the event should be ignored.
If the sink->polarity field contains both rising and falling edges, the function checks if the event should be ignored using the ts2phc_pps_sink_ignore function. If the function returns true, the event is ignored, and the function returns EXTTS_IGNORE.
If the event is not ignored, the function calculates the time stamp using the pct_to_tmv function to convert the event timestamp into a tmv_t value, which is then adjusted by the sink's correction value. The function then adds the resulting time stamp to the sink's clock using the ts2phc_clock_add_tstamp function.

The function returns EXTTS_OK to indicate that the event was handled successfully. If an error occurred or the event should be ignored, the function returns EXTTS_ERROR or EXTTS_IGNORE, respectively. Overall, this function is responsible for handling an external time stamp event on a PPS sink, including reading the event data, calculating the corresponding time stamp, and adding it to the sink's clock.

6.7.3 ts2phc_pps_source_getppstime

This function retrieves the current system time stamp from a PPS source. The function takes a struct ts2phc_pps_source pointer as an argument, which represents the PPS source. The function then calls the getppstime function pointer of the source to retrieve the current system time stamp and store it in the struct timespec pointed to by ts.

The getppstime function pointer is set to a specific function implementation when the PPS source is initialized in the ts2phc_pps_source_register function. The implementation of the getppstime function varies depending on the type of PPS source, such as a PHC clock or a PPS signal from a GPIO pin.

Overall, this function provides a convenient interface for retrieving the current system time stamp from a PPS source, abstracting away the details of the specific source implementation.

6.7.4 ts2phc_clock_add_tstamp

This function adds a time stamp to a clock by converting the time stamp value from a tmv_t type to a struct timespec type and storing it in the clock. The function is called by the ts2phc_pps_sink_event function to add the calculated time stamp to the sink's clock.

The function takes a struct ts2phc_clock pointer and a tmv_t value as arguments. The struct ts2phc_clock structure represents a clock that can receive time stamps, which contains information about the clock's name, last time stamp, and whether a time stamp is available.

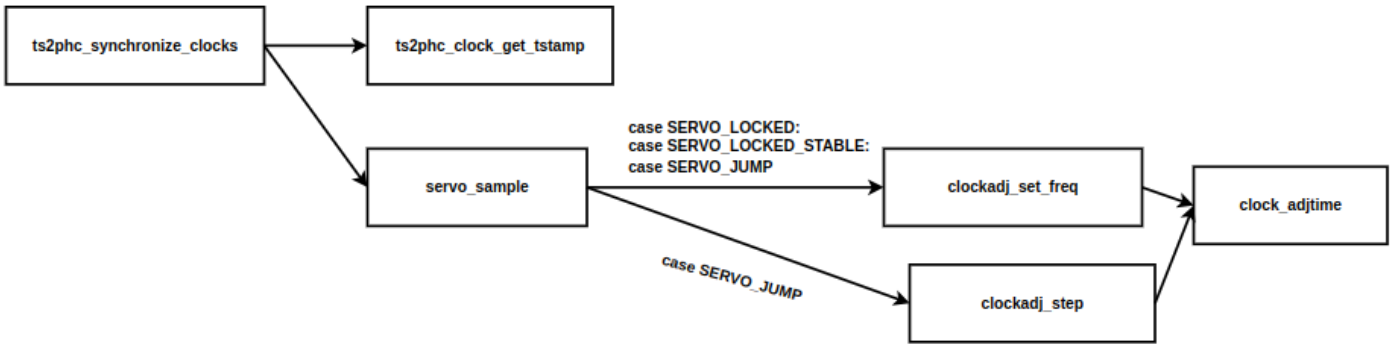
The function first converts the tmv_t value to a struct timespec value using the tmv_to_timespec function. The resulting struct timespec value represents the time stamp to be added to the clock.

The function then stores the time stamp in the clock's last_ts field and sets the is_ts_available flag to true, indicating that a time stamp is available in the clock. The function also logs a debug message indicating the time stamp value and the clock name.

Overall, this function provides a simple interface for adding a time stamp to a clock and updating the clock's status.

6.8. Synchronization

Finally, after ts2phc received the nmea message, it will conduct to synchronize the PHC with toD. This is the diagram which describe this process:



6.8.1. ts2phc_synchronize_clocks

This function synchronizes the time of a set of clocks with a reference clock or a PPS source by adjusting the frequency of the clocks using a PI controller. The function is called by the ts2phc_pps_thread function at each cycle of the main loop.

The function takes a struct ts2phc_private pointer and an autocfg flag as arguments. If autocfg is true, the function uses the reference clock specified in priv->ref_clock to synchronize the clocks. Otherwise, the function retrieves the current time stamp from the PPS source using the ts2phc_pps_source_implicit_tstamp function.

The function then iterates over all clocks in the priv->clocks list and checks if a clock is a target for synchronization using the c->is_target flag. If a clock is not a target, the function skips it.

For each target clock, the function retrieves the current time stamp using the ts2phc_clock_get_tstamp function. If the time stamp is not valid, the function skips the clock.

The function then calculates the time offset between the target clock and the reference clock or PPS source using the tmv_to_nanoseconds function. If the clock has the no_adj flag set, the function logs the offset and skips further processing for that clock.

If the clock does not have the no_adj flag set, the function calculates the adjustment value using a PI controller implemented by the servo_sample function. The controller uses the time offset and the current time stamp of the target clock, along with a set of control parameters, to calculate the adjustment value.

The function then logs the time offset and adjustment value for the target clock and uses the clockadj_set_freq and clockadj_step functions to adjust the frequency of the clock. The adjustment method depends on the state of the PI controller, which can be unlocked, locked, locked and stable, or jump. If an error occurs during the frequency adjustment, the function resets the PI controller and unlocks the clock.

Overall, this function provides a mechanism for synchronizing the time of a set of clocks with a reference clock or PPS source by adjusting their frequency using a PI controller. The function iterates over all target clocks, calculates the time offset and adjustment value, and adjusts their frequency based on the PI controller's state.

6.8.2. ts2phc_clock_get_tstamp

This function retrieves the last time stamp stored in a clock and marks the clock as unavailable for further time stamps. The function is called by the ts2phc_synchronize_clocks function to retrieve the time stamp of a clock.
The function takes a struct ts2phc_clock pointer and a pointer to a tmv_t value as arguments. The struct ts2phc_clock structure represents a clock that can receive time stamps, which contains information about the clock's name, last time stamp, and whether a time stamp is available.
The function first checks if a time stamp is available in the clock by checking the is_ts_available flag. If there is no time stamp available, the function returns 0, indicating that the time stamp is not valid.
If a time stamp is available, the function marks the clock as unavailable for further time stamps by setting the is_ts_available flag to false. The function then stores the last time stamp value in the ts pointer and returns 1, indicating that the time stamp is valid.
Overall, this function provides a simple interface for retrieving the last time stamp value stored in a clock and marking the clock as unavailable for further time stamps.

6.8.3. servo_sample

This function performs a single sample of a PI controller implemented by a struct servo structure. The PI controller is used to adjust the frequency of a clock to synchronize its time with a reference clock or PPS source. The function is called by the ts2phc_synchronize_clocks function for each target clock.
The function takes a struct servo pointer, an integer offset representing the time offset between the target clock and the reference clock or PPS source, a uint64_t value representing the local time stamp of the target clock, a double value representing the weight of the sample, and a pointer to an enum servo_state value representing the current state of the PI controller.
The function first calls the sample function pointer of the servo structure to perform a single sample of the PI controller. The sample function calculates the adjustment value based on the time offset and other parameters of the PI controller.
The function then updates the state of the PI controller based on the result of the sample. If the state is SERVO_UNLOCKED, the function sets the curr_offset_values field of the servo structure to the maximum value of the num_offset_values field, indicating that the offset values are invalid. If the state is SERVO_JUMP, the function

sets the curr_offset_values field and first_update field of the servo structure to 0, indicating that the clock has jumped in time. If the state is SERVO_LOCKED, the function checks if the offset value is within a threshold using the check_offset_threshold function and sets the state pointer to SERVO_LOCKED_STABLE if the offset is within the threshold.
Finally, the function returns the adjustment value calculated by the sample function.
Overall, this function provides an interface for performing a single sample of a PI controller implemented by a struct servo structure to adjust the frequency of a clock and synchronize its time with a reference clock or PPS source.

6.8.4. clockadj_set_freq

This function sets the frequency adjustment of a clock using the clock_adjtime system call. The function is called by the ts2phc_synchronize_clocks function to adjust the frequency of a clock based on the output of a PI controller.
The function takes a clockid_t value representing the clock identifier and a double value representing the frequency adjustment in parts per million (ppm). The clockid_t value is used to identify the clock to be adjusted, while the double value represents the frequency adjustment in ppm.
The function first initializes a struct timex variable tx to zero using the memset function. The struct timex structure is used to pass information to the clock_adjtime system call.
If the clock identifier is CLOCK_REALTIME and the realtime_nominal_tick variable is set, the function calculates the tick adjustment value and sets the ADJ_TICK flag in the modes field of the tx variable. The tick adjustment value is calculated based on the freq value and the nominal tick length of the clock, and is added to the tick field of the tx variable. The realtime_hz variable represents the frequency of the clock in Hz.
The function then sets the ADJ_FREQUENCY flag in the modes field of the tx variable and calculates the frequency adjustment value in ppb (parts per billion) by multiplying the freq value by 65.536. The frequency adjustment value is then cast to a long and stored in the freq field of the tx variable.
Finally, the function calls the clock_adjtime system call with the clock identifier and the tx variable as arguments. If the system call fails, the function logs an error message and returns -1. Otherwise, the function returns 0 to indicate success.
Overall, this function provides an interface for setting the frequency adjustment of a clock using the clock_adjtime system call. The function calculates the tick and frequency adjustment values based on the input parameters and sets the appropriate flags in the tx variable before calling the system call.

6.8.5. clockadj_step

This function steps the time of a clock by a specified amount using the clock_adjtime system call. The function is called by the ts2phc_synchronize_clocks function to adjust the time of a clock based on the output of a PI controller.

The function takes a clockid_t value representing the clock identifier and an int64_t value representing the amount of time to step the clock in nanoseconds. The clockid_t value is used to identify the clock to be stepped, while the int64_t value represents the amount of time to step the clock in nanoseconds.

The function first initializes a struct timex variable tx to zero using the memset function. The struct timex structure is used to pass information to the clock_adjtime system call.

The function then checks if the step value is negative, and if so, sets the sign variable to -1 and takes the absolute value of the step value. The sign variable is used to indicate the direction of the step.

The function sets the ADJ_SETOFFSET and ADJ_NANO flags in the modes field of the tx variable to indicate that the time offset is being set in nanoseconds. The function then sets the tv_sec and tv_usec fields of the time field of the tx variable based on the sign and value of the step parameter. If the tv_usec field is negative, the function adjusts the tv_sec and tv_usec fields to ensure that tv_usec is non-negative.

Finally, the function calls the clock_adjtime system call with the clock identifier and the tx variable as arguments. If the system call fails, the function logs an error message and returns -1. Otherwise, the function returns 0 to indicate success.

Overall, this function provides an interface for stepping the time of a clock by a specified amount using the clock_adjtime system call. The function calculates the time offset based on the input parameters and sets the appropriate flags in the tx variable before calling the system call.

6.8.6. clock_adjtime

This function is a wrapper for the clock_adjtime system call, which is used to adjust the time and frequency of a clock. The function takes a clockid_t value representing the clock identifier and a pointer to a struct timex variable tx, which contains information about the time and frequency adjustment to be made.

The function calls the syscall function with the __NR_clock_adjtime system call number, the clock identifier id, and a pointer to the tx variable as arguments. The syscall function is a low-level interface to make system calls in Linux.

The clock_adjtime system call adjusts the time and/or frequency of the specified clock based on the information in the tx variable. The tx variable contains a set of flags indicating the type of adjustment to be made, as well as values for the time and/or frequency adjustments.

If the system call is successful, the function returns 0. Otherwise, the function returns -1 to indicate an error.
Overall, this function provides a simple interface for making the clock_adjtime system call to adjust the time and frequency of a clock.