Bitbus Protocol Verification Properties

Formal Specification and FSM → C Implementation Utility

This document presents the complete set of formal verification properties for the Bitbus protocol, organised by functional category. For each property we describe its temporal-logic formula, its role in system validation, and its specific utility for implementing a Finite State Machine (FSM) in embedded C code.

D.1 Structural Safety Properties

Property	P1
Formula	`A[] not deadlock`
Function	Verifies the absence of global deadlock in the system of master and slave automata. Guarantees that no system configuration reaches a state where no transition can fire, ensuring continuous protocol progress.
FSM→C Utility	*Essential for C implementation: ensures no state combination can block the code indefinitely. Guides event-loop design so every state always has at least one outgoing transition. Identifies states that need a timeout or watchdog to prevent blocking when expected messages are not received.*

Property	P2
Formula	`A[] ((MASTER_SDLC.slave_state_in_m == NRM and MASTER_BITBUS.irrec_err == 0) imply not deadlock)`
Function	Verifies absence of deadlock specifically in nominal mode (NRM) when the SDLC connection is established and no irrecoverable error exists. Ensures that normal protocol operation never leads to a deadlock.
FSM→C Utility	*Crucial for nominal-path code: confirms that main execution paths (no errors) are always non-blocking. Lets production code clearly separate nominal logic (guaranteed deadlock-free) from error handling, justifying the omission of some timeouts in nominal states.*

D.2 Data Integrity and Protocol Validity

Property	P3
Formula	`MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s --> MASTER_BITBUS.PROCESS_BITBUS`
Function	Verifies that when the master receives an alphanumeric slave frame with a valid checksum (cs_frame_s == cs_ref_s), it always transitions to PROCESS_BITBUS. Guarantees correct handling of valid frames.
FSM→C Utility	*Dictates checksum-validation implementation: the C checksum function must be called immediately after reception; if valid, trigger an unconditional transition to PROCESS_BITBUS. No additional intermediate check is needed. Pattern: if (checksum_valid(frame)) { state = PROCESS_BITBUS; }*

Property	P4

Formula	`E<> MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s`
Function	Verifies reachability of the state where the master receives a valid alphanumeric frame. Confirms at least one execution path leads to valid-frame reception, proving the nominal scenario is possible.
FSM→C Utility	*Validates code testability: guarantees a unit-test scenario can be built where a valid frame is received. Foundation for integration tests and nominal test cases. This path must be covered by code-coverage tests.*

Property	**P5**
Formula	`(MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s) --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST`
Function	Verifies that on receiving an alphanumeric frame with invalid checksum, the master always goes to SDLC_CONNECTION_REQUEST. Guarantees detection and proper handling of corrupted frames.
FSM→C Utility	*Determines checksum-error handling: on invalid checksum the C implementation must immediately reset to SDLC_CONNECTION_REQUEST without local recovery. Simplifies code by eliminating retry logic at this level. Pattern: if (!checksum_valid(frame)) { state = SDLC_CONNECTION_REQUEST; reset_counters(); }*

Property	**P6**
Formula	`E<> (MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s)`
Function	Verifies reachability of the state where the master receives an alphanumeric frame with invalid checksum. Confirms the transmission-error scenario is represented in the model.
FSM→C Utility	*Guides robustness-test implementation: confirms that tests with injected corrupted frames must be created. Verifies that error paths are reachable and must be tested. Justifies data-corruption tests in the validation suite.*

Property	**P7**
Formula	`(SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.SEND_ALPHANUM_S`
Function	Verifies that when the slave receives a valid master data frame (correct checksum) and the response delay T_rep is reached, it always transitions to the alphanumeric-response sending state. Guarantees response-timing compliance for valid frames.
FSM→C Utility	*Imposes strict temporal synchronisation: the C implementation must use a timer (y) that reaches exactly T_rep before sending. Pattern: if (checksum_valid && timer_y >= T_rep) { state = SEND_ALPHANUM_S; } Prevents premature responses and guarantees deterministic timing.*

Property	**P8**
Formula	`E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep)`
Function	Verifies reachability of the state where the slave receives a valid data frame and reaches the response delay. Confirms the nominal slave→master scenario is possible.

FSM→C Utility	*Validates testability of the nominal slave path: confirms that correct timing can be simulated. Guides implementation of timing tests with T_rep timer mocks. This scenario must be covered in temporal integration tests.*

Property	**P9**
Formula	`(SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.ERROR_IN_S`
Function	Verifies that when the slave receives a data frame with invalid checksum and T_rep is reached, it always transitions to ERROR_IN_S. Guarantees transmission-error detection on the slave side.
FSM→C Utility	*Specifies timed error behaviour: even on invalid checksum the slave waits T_rep before reporting. Avoids transitions too fast that could disturb the protocol. Pattern: if (!checksum_valid && timer_y >= T_rep) { state = ERROR_IN_S; log_error(); }*

Property	**P10**
Formula	`E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep)`
Function	Verifies reachability of the state where the slave receives an invalid data frame after the response delay. Confirms the slave-side error scenario is modelled.
FSM→C Utility	*Guides timed error tests: confirms behaviour with a corrupted frame after waiting for the delay can be tested. Justifies tests combining data corruption with temporal constraint compliance.*

Property	**P11**
Formula	`(SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m) --> (SLAVE_BITBUS.SEND_DATA_RESPONSE and invalid_data_m == 1)`
Function	Verifies that when the slave receives a frame with invalid format but valid checksum, it sends a data response with the invalid_data_m flag set to 1. Guarantees format-error signalling.
FSM→C Utility	*Distinguishes two validation levels: the C code must perform two sequential checks. If format is invalid but checksum OK, the slave responds preserving the invalid-frame information: if (checksum_valid && invalid_data_m) { response.invalid_data = 1; state = SEND_DATA_RESPONSE; }*

Property	**P12**
Formula	`E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m)`
Function	Verifies reachability of the state where the slave receives a frame with invalid format but correct checksum. Confirms this protocol-error scenario is represented.
FSM→C Utility	*Guides protocol-validation tests: tests with syntactically invalid frames but correct checksum must be created. Validates coverage of high-level protocol errors.*

Property	**P13**
Formula	`(SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m) --> SLAVE_BITBUS.ERROR_IN_S`

Function	Verifies that when the slave receives a frame with both invalid format AND incorrect checksum, it always transitions to the error state. Guarantees strict handling of combined errors.
FSM→C Utility	*Specifies multiple-error handling: combined errors trigger a direct transition to ERROR_IN_S. Simplifies code by avoiding separate handling. Pattern: if (!checksum_valid || !format_valid) { state = ERROR_IN_S; } with checksum taking priority.*

Property	**P14**
Formula	`E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m)`
Function	Verifies reachability of the state where the slave receives a frame with multiple errors (format and checksum). Confirms complete representation of error scenarios.
FSM→C Utility	*Justifies worst-case tests: validates that frames with several simultaneous error types can be tested. Guides implementation of extreme robustness tests.*

Property	**P15**
Formula	`(MASTER_BITBUS.RECEIVE_VALID_FRM and cs_frame_s == cs_ref_s) --> MASTER_BITBUS.PROCESS_BITBUS`
Function	Verifies that when the master receives a valid frame (correct format) with correct checksum, it always transitions to PROCESS_BITBUS. Guarantees proper handling of protocol-compliant frames.
FSM→C Utility	*Confirms complete master-side validation logic: both checksum AND format must be validated before processing. Structures reception code with two-step validation. Pattern: if (checksum_valid(frame) && format_valid(frame)) { state = PROCESS_BITBUS; } otherwise error handling.*

Property	**P16**
Formula	`E<> (MASTER_BITBUS.RECEIVE_DATA_FRAME and cs_frame_s == cs_ref_s)`
Function	Verifies reachability of the state where the master receives a valid data frame with correct checksum. Confirms that slave→master communication can be performed correctly.
FSM→C Utility	*Validates bidirectionality: confirms the master reception path is functional. Guides implementation of bidirectional communication tests.*

D.3 Temporal Constraints

Property	**P17**
Formula	`A[] (MASTER_BITBUS.SEND_PROCESS imply MASTER_BITBUS.x == T_Pol)`
Function	Verifies that every time the master reaches SEND_PROCESS, its clock x equals exactly T_Pol (polling period). Guarantees strict compliance with the cyclic polling timing defined by the industrial protocol.
FSM→C Utility	*Imposes strict periodic timing: the C implementation must use a precise cyclic timer (x). Transmission may only occur when x == T_Pol. Pattern: if (timer_x >= T_Pol) { state = SEND_PROCESS; } Justifies use of high-precision timers.*

Property	P18
Formula	`(MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == T_out_M) --> (MASTER_BITBUS.PROCESS_BITBUS or MASTER_BITBUS.SDLC_CONNECTION_REQUEST)`
Function	Verifies that when the master is waiting for a Bitbus response and timeout T_out_M expires, it transitions either to PROCESS_BITBUS (response received in time) or to SDLC_CONNECTION_REQUEST. No timeout leaves the system in indefinite wait.
FSM→C Utility	*Specifies master timeout handling: at T_out_M expiry, two transitions depending on whether a response was received. Pattern: if (timer_x >= T_out_M) { state = response_received ? PROCESS_BITBUS : SDLC_CONNECTION_REQUEST; } Prevents infinite waits.*

Property	P19
Formula	`E<> (MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == T_out_M)`
Function	Verifies reachability of the state where the master reaches its timeout while waiting for a response. Confirms the master-timeout scenario is modelled and reachable.
FSM→C Utility	*Guides timeout tests: validates that a timeout can be simulated by withholding a response for T_out_M. Justifies non-response and slave-disconnect tests.*

Property	P20
Formula	`(SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S) --> SLAVE_BITBUS.T_OUT_EXCEEDED_S`
Function	Verifies that when the slave clock y reaches or exceeds T_out_S, the slave always transitions to T_OUT_EXCEEDED_S. Guarantees detection and handling of slave-side timeouts.
FSM→C Utility	*Defines slave timeout: unconditional transition to T_OUT_EXCEEDED_S when y >= T_out_S. Pattern: if (timer_y >= T_out_S) { state = T_OUT_EXCEEDED_S; } No complex recovery logic needed at this point — just detect and transition.*

Property	P21
Formula	`E<> (SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S)`
Function	Verifies reachability of the state where the slave reaches its timeout. Confirms the slave-timeout scenario is represented in the model.
FSM→C Utility	*Guides slave timeout tests: validates that an absence of master communication can be simulated. Justifies master-loss and autonomous slave-recovery tests.*

Property	P22
Formula	`A[] (SLAVE_BITBUS.SEND_LINK_RESP imply SLAVE_BITBUS.y == T_rep)`
Function	Verifies that every time the slave sends a LINK_RESP, its clock y equals exactly T_rep. Guarantees response-timing compliance for link requests.

FSM→C Utility	*Imposes a fixed response delay for LINK_RESP: the slave must wait exactly T_rep before sending. Pattern: wait_until(timer_y == T_rep); send_link_response(); timer_y = 0; Guarantees predictable and protocol-compliant temporal behaviour.*

Property	**P23**
Formula	`A[] (SLAVE_BITBUS.SEND_UNLINK_RESP imply SLAVE_BITBUS.y == T_rep)`
Function	Verifies that every time the slave sends an UNLINK_RESP, its clock y equals T_rep. Guarantees response-timing compliance for unlink requests.
FSM→C Utility	*Unifies timing of all responses: UNLINK_RESP uses the same T_rep delay as LINK_RESP. Simplifies implementation by using a generic response-delay function. Reusable pattern: response_delay(T_rep); send_unlink_response();*

Property	**P24**
Formula	`A[] (SLAVE_BITBUS.SEND_DATA_RESPONSE imply SLAVE_BITBUS.y == T_rep)`
Function	Verifies that every time the slave sends a data response, its clock y equals T_rep. Guarantees response-timing compliance for all data requests.
FSM→C Utility	*Confirms temporal uniformity: all slave responses (LINK, UNLINK, DATA) use T_rep. A single timing-management function can be implemented for all responses, reducing complexity and error risk.*

D.4 Liveness Properties

Property	**P25**
Formula	`MASTER_BITBUS.PROCESS_BITBUS --> MASTER_BITBUS.SEND_PROCESS`
Function	Verifies that every time the master reaches PROCESS_BITBUS, it will eventually reach SEND_PROCESS. Guarantees absence of livelock and continuous communication cycle progress.
FSM→C Utility	*Guarantees code progress: PROCESS_BITBUS cannot be a terminal state. The C implementation must ensure a transition to SEND_PROCESS always follows processing. Pattern: process_data(); state = SEND_PROCESS; Infinite loops in PROCESS_BITBUS are forbidden.*

Property	**P26**
Formula	`MASTER_BITBUS.WAIT_FOR_CONNECTION and SLAVE_SDLC.ACK --> MASTER_BITBUS.PROCESS_BITBUS`
Function	Verifies that when the master waits for a connection and the SDLC slave sends an ACK, the master will eventually reach PROCESS_BITBUS. Guarantees correct Bitbus resumption after SDLC connection establishment.
FSM→C Utility	*Specifies cross-layer synchronisation: after receiving an SDLC ACK, transition to PROCESS_BITBUS to resume Bitbus. Pattern: if (state == WAIT_FOR_CONNECTION && sdlc_ack_received()) { state = PROCESS_BITBUS; } Ensures SDLC-Bitbus layer consistency.*

D.5 Resource Bounds

Property	P27
Formula	`A[] (MASTER_BITBUS.attempt_bb <= 2)`
Function	Verifies that the Bitbus retry counter (attempt_bb) never exceeds 2. Guarantees an upper bound on retransmissions to prevent infinite retry loops and ensure deterministic resource use.
FSM→C Utility	*Bounds Bitbus retries: the C code must implement an attempt_bb counter with strict check <= 2. Pattern: if (++attempt_bb > 2) { abandon_and_reset(); } else { retry_transmission(); } Guarantees deterministic termination and avoids resource exhaustion on persistent failure.*

Property	P28
Formula	`A[] (MASTER_BITBUS.attempt_link <= 3)`
Function	Verifies that the link attempt counter (attempt_link) never exceeds 3. Guarantees an upper bound on connection attempts to prevent infinite retries on persistent slave failure.
FSM→C Utility	*Bounds link attempts: maximum 3 LINK tries. Pattern: if (++attempt_link > 3) { report_link_failure(); state = ERROR_STATE; } else { resend_link_request(); } Enables rapid slave-failure detection and supervisory system alerting.*

Property	P29
Formula	`A[] (MASTER_BITBUS.attempt_unlink <= 3)`
Function	Verifies that the unlink attempt counter (attempt_unlink) never exceeds 3. Guarantees an upper bound on disconnection attempts, ensuring deterministic termination even when the slave does not respond.
FSM→C Utility	*Bounds unlink attempts: maximum 3 UNLINK tries. Pattern: if (++attempt_unlink > 3) { force_disconnect(); } else { resend_unlink_request(); } Allows forced disconnection after 3 failures, preventing the master from waiting indefinitely for unlink confirmation.*

D.6 Sequencing with the SDLC Layer

Property	P30
Formula	`A[] (MASTER_BITBUS.SEND_LINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM)`
Function	Verifies that every time the Bitbus master sends a link request, the SDLC slave state seen by the master is NRM (Normal Response Mode). Guarantees Bitbus link operations are only initiated when the SDLC layer is stable and operational.
FSM→C Utility	*Imposes an SDLC precondition: before sending LINK_REQ, verify slave_state == NRM. Pattern: if (slave_sdlc_state != NRM) { wait_for_sdlc_ready(); } send_link_request(); Ensures protocol-layer consistency and prevents Bitbus requests over an unestablished SDLC link.*

Property	P31

Formula	`A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply MASTER_SDLC.slave_state_in_m == NRM)`
Function	Verifies that every time the master sends alphanumeric data, the SDLC state is NRM. Guarantees application data exchanges only occur on an established, stable SDLC connection.
FSM→C Utility	*Precondition for data transmission: verify slave_state == NRM before every data send. Pattern: assert(slave_sdlc_state == NRM); send_alphanum_data(); Prevents data loss by ensuring the link layer is operational.*

Property	**P32**
Formula	`A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM)`
Function	Verifies that every time the master sends an unlink request, the SDLC state is NRM. Guarantees unlink operations are performed in the context of an active SDLC connection.
FSM→C Utility	*Precondition for unlinking: SDLC must be in NRM. Pattern: if (slave_sdlc_state == NRM) { send_unlink_request(); } else { log_error("SDLC not ready"); } Ensures clean session termination on an active link.*

Property	**P33**
Formula	`A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply !SLAVE_NO_LINKED)`
Function	Verifies that every time the master sends alphanumeric data, the slave is linked (SLAVE_NO_LINKED == 0). Guarantees no application data exchange occurs without a prior Bitbus link.
FSM→C Utility	*Double precondition: SDLC in NRM AND Bitbus linked. Pattern: if (slave_sdlc_state == NRM && slave_linked) { send_data(); } else { error_not_linked(); } Strengthens safety by checking both protocol layers before transmission.*

Property	**P34**
Formula	`A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply !SLAVE_NO_LINKED)`
Function	Verifies that every time the master sends an unlink request, the slave is already linked. Guarantees logical consistency by forbidding unlink attempts on a non-existent connection.
FSM→C Utility	*Prevents invalid UNLINKs: only unlink if already linked. Pattern: if (!slave_linked) { log_error("Already unlinked"); return; } send_unlink_request(); Prevents logic errors and inconsistent states.*

D.7 Error Handling and Reset

Property	**P35**
Formula	`MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST`

Function	Verifies that every time the master detects an error (ERROR_IN_M), it will eventually reach the reset state (SDLC_CONNECTION_REQUEST). Guarantees every master-side error leads to a controlled protocol reinitialisation.
FSM→C Utility	*Error-recovery strategy: every master error triggers a full reset to SDLC_CONNECTION_REQUEST. Pattern: if (error_detected) { cleanup_state(); state = SDLC_CONNECTION_REQUEST; } Simplifies error handling using a global reset strategy rather than complex local recovery.*

Property	**P36**
Formula	`SLAVE_BITBUS.ERROR_IN_S --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST`
Function	Verifies that every time the slave detects an error (ERROR_IN_S), the master will eventually reach the reset state. Guarantees propagation of slave errors to the master and synchronised error recovery.
FSM→C Utility	*Master-slave error synchronisation: a slave error must be signalled to the master, which initiates the reset. Requires an error-signalling mechanism (flag, special message). Master code: if (slave_error_received) { state = SDLC_CONNECTION_REQUEST; } Ensures coordinated recovery.*

Property	**P37**
Formula	`A[] (MASTER_BITBUS.ERROR_IN_M imply MASTER_SDLC.slave_state_in_m == NRM)`
Function	Verifies that every time the master is in error state, the underlying SDLC state is NRM. Guarantees Bitbus errors only occur in the context of an established SDLC connection.
FSM→C Utility	*Distinguishes Bitbus vs SDLC errors: a Bitbus error occurs on an operational SDLC. Guides diagnostics: if (error && sdlc_state != NRM) { root_cause = SDLC_LAYER; } else { root_cause = BITBUS_LAYER; }*

Property	**P38**
Formula	`A[] (SLAVE_BITBUS.ERROR_IN_S imply SLAVE_SDLC.slave_state == NRM)`
Function	Verifies that every time the slave is in error state, the slave SDLC state is NRM. Guarantees the same consistency on the slave side: application errors are detected on a functional link layer.
FSM→C Utility	*Same principle on the slave side: Bitbus error only if SDLC is in NRM. Simplifies error diagnosis and fault localisation in the layered architecture.*

D.8 Reachability of Key States

Property	**P39**
Formula	`E<> MASTER_BITBUS.PROC_SEND_LINK_REQ`
Function	Verifies reachability of the state where the master processes sending a link request. Confirms the link phase is effectively accessible in the model, validating completeness of the connection-establishment scenario.

FSM→C Utility	*Validates completeness of link code: confirms a path to PROC_SEND_LINK_REQ exists. Justifies connection-establishment tests and guarantees this feature is implemented and reachable.*

Property	**P40**
Formula	`E<> MASTER_BITBUS.PROC_SEND_UNLINK_REQ`
Function	Verifies reachability of the state where the master processes sending an unlink request. Confirms the graceful connection-termination phase is modelled and accessible.
FSM→C Utility	*Validates completeness of unlink code: confirms the graceful disconnection path is implemented. Guides session-termination and resource-cleanup tests.*

Property	**P41**
Formula	`E<> MASTER_BITBUS.PROC_SEND_ALPHANUM_M`
Function	Verifies reachability of the state where the master processes sending alphanumeric data. Confirms the application data exchange phase is accessible, validating the nominal communication scenario.
FSM→C Utility	*Validates the complete nominal path: from initialisation to application data sending. Confirms the implementation can reach the protocol's main objective (data transfer). Justifies end-to-end tests.*

Property	**P42**
Formula	`E<> SLAVE_BITBUS.SEND_LINK_RESP`
Function	Verifies reachability of the state where the slave sends a link response. Confirms the slave can respond to link requests, validating reactive slave behaviour for connection establishment.
FSM→C Utility	*Validates slave reactivity: confirms the slave can process and respond to LINK_REQ. Guides bidirectional handshake and master-slave synchronisation tests.*

Property	**P43**
Formula	`E<> SLAVE_BITBUS.SEND_UNLINK_RESP`
Function	Verifies reachability of the state where the slave sends an unlink response. Confirms the slave can participate in graceful connection termination.
FSM→C Utility	*Validates bidirectional cleanup: confirms the slave actively participates in termination. Guides clean disconnection and slave-side resource-release tests.*

Property	**P44**
Formula	`E<> SLAVE_BITBUS.SEND_ALPHANUM_S`
Function	Verifies reachability of the state where the slave sends alphanumeric data. Confirms the slave can emit data responses, validating full bidirectional communication.
FSM→C Utility	*Validates bidirectional communication: confirms the slave can not only receive but also send data. Justifies full-duplex and two-way data transfer tests.*

D.9 Logical Consistency, Mutual Exclusion and Traceability

Property	P45
Formula	`A[] (MASTER_BITBUS.frame_sent_m == data_response_m imply SLAVE_NO_LINKED == 0)`
Function	Verifies that every time a data-response frame is sent, the slave is linked (SLAVE_NO_LINKED == 0). Guarantees data responses are only emitted within an established Bitbus session.
FSM→C Utility	*Session invariant: never send data without an active link. Pattern: assert(slave_linked == true); send_data_response(); Prevents logic errors where data would be sent outside a session context.*

Property	P46
Formula	`A[] (MASTER_BITBUS.SEND_LINK_REQ imply SLAVE_NO_LINKED == 1)`
Function	Verifies that every time the master sends a link request, the slave is not linked (SLAVE_NO_LINKED == 1). Guarantees logical consistency by forbidding redundant link requests on an already-established session.
FSM→C Utility	*Prevents redundant LINKs: only link if not already linked. Pattern: if (slave_linked) { log_error("Already linked"); return; } send_link_request(); Prevents state inconsistencies and avoids unnecessary operations.*

Property	P47
Formula	`A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply SLAVE_NO_LINKED == 0)`
Function	Verifies that every time the master sends an unlink request, the slave is linked. Guarantees unlink is only attempted on existing connections, maintaining session-state consistency.
FSM→C Utility	*Unlink consistency: only unlink if linked (see P34). Reinforces with a strict precondition in code.*

Property	P48
Formula	`MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.PROCESS_BITBUS`
Function	Verifies that every time the master detects an error, it will eventually return to PROCESS_BITBUS. Guarantees that after error handling and reset, the system returns to a stable operational state.
FSM→C Utility	*Guarantees recovery: after error and reset, return to operational state (PROCESS_BITBUS). Ensures the code never stays permanently in an error state. Recovery pattern: error_handler() -> reset() -> state = PROCESS_BITBUS; resume_normal_operation();*

Property	P49
Formula	`A[] not (MASTER_BITBUS.SEND_LINK_REQ and MASTER_BITBUS.SEND_UNLINK_REQ)`

Function	Verifies strict mutual exclusion between sending link requests and unlink requests. Guarantees no system state allows both contradictory operations simultaneously.
FSM→C Utility	*LINK/UNLINK mutual exclusion: implemented via mutually exclusive states. With a single state variable (enum), mutual exclusion is guaranteed naturally. No additional locks required.*

Property	**P50**
Formula	`A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_LINK_REQ)`
Function	Verifies mutual exclusion between sending alphanumeric data and link requests. Guarantees the link phase and data transfer phase do not overlap.
FSM→C Utility	*LINK-then-DATA sequencing: never simultaneous. Strict sequential implementation: complete LINK before allowing DATA. Pattern: state = SEND_LINK_REQ; wait_link_complete(); state = SEND_ALPHANUM_M;*

Property	**P51**
Formula	`A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_UNLINK_REQ)`
Function	Verifies mutual exclusion between sending alphanumeric data and unlink requests. Guarantees data transfers and session termination are sequential, not concurrent.
FSM→C Utility	*DATA-then-UNLINK sequencing: complete transfers before unlinking. Pattern: finish_data_transfer(); state = SEND_UNLINK_REQ; Prevents loss of in-transit data during disconnection.*

Property	**P52**
Formula	`A[] not ((SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_UNLINK_RESP) or (SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE) or (SLAVE_BITBUS.SEND_UNLINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE))`
Function	Verifies complete mutual exclusion among the three slave response types (link, unlink, data). Guarantees the slave sends only one response type at a time, ensuring exchange consistency and non-ambiguity.
FSM→C Utility	*Ternary mutual exclusion: one response type at a time. Implementation: enum ResponseType { LINK_RESP, UNLINK_RESP, DATA_RESP }; ResponseType current_response; Simplifies master-side reception code, which handles only one response type per cycle.*

Property	**P53**
Formula	`MASTER_BITBUS.SEND_LINK_REQ --> (MASTER_BITBUS.RECEIVE_LINK_RESP or MASTER_BITBUS.ERROR_IN_M or MASTER_BITBUS.SDLC_CONNECTION_REQUEST)`
Function	Verifies request-response traceability for linking: after a link request is sent, the system will reach either response reception, an error state, or a new SDLC connection request. Guarantees no link request is left without a defined outcome.
FSM→C Utility	*Guarantees a deterministic future after LINK_REQ: three clearly-defined outcomes. Guides timeout and error-handling implementation: after SEND_LINK_REQ, wait for*

	RECEIVE_LINK_RESP with a timeout leading to ERROR_IN_M or SDLC_CONNECTION_REQUEST. Pattern: send_link_req(); wait_response_or_timeout(); handle_outcome();

Property	**P54**

Formula	`MASTER_BITBUS.SEND_UNLINK_REQ -->`
	`(MASTER_BITBUS.RECEIVE_UNLINK_RESP or`
	`MASTER_BITBUS.SDLC_CONNECTION_REQUEST)`

Function	Verifies request-response traceability for unlinking: after an unlink request is sent, the system will reach either an unlink response or a new SDLC connection request. Guarantees no unlink request leaves the system in an undefined state.
FSM→C Utility	*Guarantees two outcomes after UNLINK_REQ: response or reset. Pattern: send_unlink_req(); wait_response_or_timeout(); if (timeout) { state = SDLC_CONNECTION_REQUEST; } else { state = RECEIVE_UNLINK_RESP; } Ensures clean termination even without slave confirmation.*

Conclusion

The 54 properties presented in this document constitute a comprehensive formal verification framework for the Bitbus protocol. Each property has been enriched with a description of its practical utility for embedded C implementation, establishing a direct bridge between formal model-checking verification and software development.

This approach guarantees that the implemented code respects not only the functional specifications, but also the safety, liveness, temporal, and logical consistency properties validated by formal verification. The constraints extracted from these properties directly guide implementation decisions regarding timer management, retry counters, state mutual exclusion, and error handling.

Together these properties form a complete methodology for developing critical systems based on formal verification, where every assertion of the formal model translates into a verifiable constraint in C code — ensuring traceability and consistency throughout the development process from specification to implementation.