GTE Government Systems

NASA - SCAR

Interim Service ISDN Satellite (ISIS) Hardware Experiment Development

for
Advanced ISDN Satellite Designs
and Experiments

30 April 1992

Task Completion Report
NASA SCAR Contract NASW-4520, 13 Sep 1990

Prepared by
Gerard R. Pepin
GTE/Government Systems
15000 Conference Center Drive
Chantilly, Virginia 22021-3808

NASA
National Aeronautics and Space Administration

Report Documentation Page

85439

1. Report No.	2. Government Accession No:	3. Recipient's Catalog No. P-171

4. Title and Subtitle
Interim Service ISDN Satellite (ISIS) Hardware Experiment Development for Advanced Satellite Designs and Experiments (Task Completion Report)

5. Report Date
30 April 1992

6. Performing Organization Code

7. Author(s)
Gerard R. Pepin

8. Performing Organization Report No.

10. Work Unit No.

9. Performing Organization Name and Address
GTE Government Systems
15000 Conference Center Drive
P.O. Box 10814
Chantilly, VA 22021-3808

11. Contract or Grant No.
NASW-4520

13. Type of Report and Period Covered
ISIS Hardware Experiment Development Task Completion Report (1 OCT 91 - 30 APR 92)

12. Sponsoring Agency Name and Address
NASA Headquarters
Headquarters Acquisition Division
300 7th Street, SW
Washington, DC 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract
The Interim Service ISDN Satellite (ISIS) Hardware Experiment Development for Advanced Satellite Designs describes the development of the ISDN Satellite Terminal Adapter (ISTA) capable of translating ISDN protocol traffic into TDMA signals for use by a communications satellite. The ISTA connects the NT1 via the U-interface on the line termination side of the CPE to the RS-499 interface for satellite uplink. The same ISTA converts in the opposite direction the RS-499 to U interface data with a simple switch setting.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
ISDN, satellite, traffic network, simulation, ISDN standards, B-ISDN, frame relay, on-orbit switching, computer networks, satellite orbits, satellite transmission quality, network configuration, traffic model	Unclassified-Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified		

SECTION 1

INTRODUCTION

1.1 Background

The objectives of this element of the NASA Satellite Communications Applications Research (SCAR) Program are to develop new advanced on-board satellite capabilities that will enable the provision of new services, namely interim and full Integrated Services Digital Network (ISDN) services via satellite and to provide a system analysis of futuristic satellite communications concepts, namely broadband services via satellite.

This aspect of the NASA SCAR Program provides a research and development effort to:

1) develop basic technologies and concepts to use the on-board processing and switching capabilities of advanced satellites that will enable the provision of interim and full ISDN services and

2) provide a systems and requirements analysis of future satellite communications concepts based on a new generation of broadband switching and processing satellites.

These objectives will be achieved in part by designing and developing hardware to interface between terrestrial ISDN networks and a communications satellite, possibly with the Advanced Communications Technology Satellite (ACTS).

1.2 Scope

This task completion report documents the ISDN Satellite Terminal Adapter (ISTA) development associated with the Interim Service ISDN Satellite (ISIS) architecture. The process and methodology is applicable to the ISIS system as described in Figure 1.2-1, "NASA/SCAR Approaches for Advanced ISDN Satellites". The ISIS Network development represents satellite systems like the Advanced Communications Technology Satellite (ACTS) orbiting switch.

The ACTS will be controlled by a Master Ground Station (MGS) shown in Figure 1.2-2, "Closed User-Oriented Scenario". A user of the ACTS satellite orbiting switch requests services from the MGS, a combination of the NASA Ground Station (NGS) and the Master Control Station (MCS). The MGS, in turn, commands the satellite to switch the appropriate communications channels.

The ultimate aim of this element of the SCAR Program is to move these MGS functions on-board the next generation ISDN communications satellite as shown in Figure 1.2-3, "Advanced ISDN Satellite". The technical and operational parameters for the advanced ISDN communications satellite design will be obtained from a software engineering model of the major subsystems of the ISDN communications satellite architecture. Discrete event simulation experiments will be performed with the model using various traffic scenarios, design parameters, and operational procedures. The data from these simulations will be analyzed using the NASA SCAR performance measures discussed in previous reports. Data from hardware experiments will used to verify the model results.

GTE Government Systems

NASA - SCAR

ISIS Interim Service ISDN Satellite	FSIS Full Service ISDN Satellite	BSIS Broadband Service ISDN Satellite
• ACTS-like Satellite Design and Transponder	• New ISDN Satellite Design with onboard Class 5 Switch and SS7 Network Interface	• Advanced ISDN Satellite Design with onboard Class 5 Switch and SS7 Network Interface and layered protocol
• Provide Narrowband ISDN Services (Basic Rate Access)	• Provide Narrowband ISDN Services (Basic/Primary Rate Access)	• Provide Broadband ISDN Services (Primary Rate Access)
• Provide remote access ISDN Satellite Terminals using ISDN Satellite Terminal Adapter	• Provide nationwide single hop single CONUS earth coverage antenna satellite link connectivity to an interexchange node for ISDN Satellite Terminals (up to 10,000 ISAT)	• Provide nationwide single hop, multiple high gain hopping beams, forward error control, optical processing, and "zero delay" satellite link interexchange node connectivity
• Will use D channel signaling but NOT SS7	• Will use D channel signaling with SS7	• Will use D channel signaling with SS7
• Will use ACTS call control and Baseband Switching Architecture	• Will use SS7 call control with minimum call set-up time and efficient satellite BW utilization	• Will center design around ATM fast packet switching techniques

Figure 1.2-1 NASA/SCAR Approaches for Advanced ISDN Satellites

5 T03B SCA DAT
Approach Adv ISDN Satel
February 4, 1992

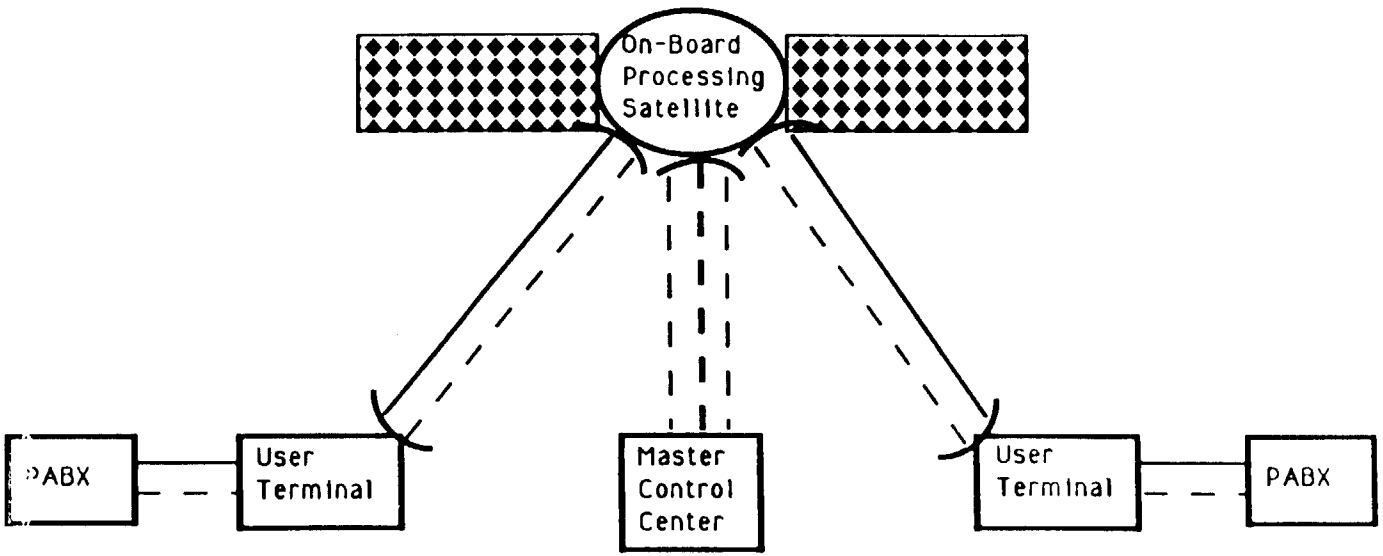


Figure 1.2 -2 Closed User-Oriented Scenario

Ref. Fournon 131SS

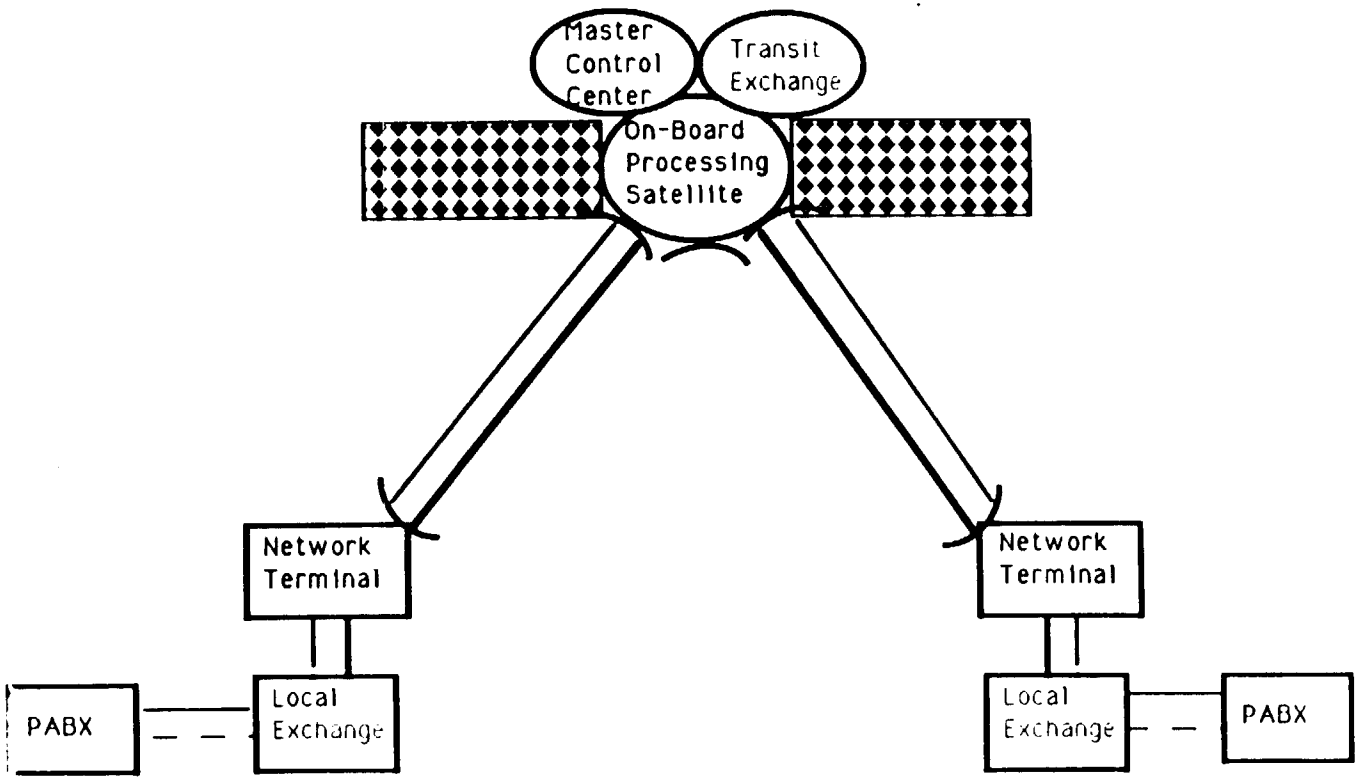


Figure 1.2-3 Advanced ISDN Satellite

In order to associate modeling and simulation results with real-world data, some ISDN hardware design and development were undertaken. Hardware development was limited to the ISIS approach. Figure 1.2-4, "ISIS System Configuration for Remote Access", illustrates the ISIS system configuration. The ISDN Satellite Terminal Adapter (ISTA) hardware was designed to interface with the Type 1 network termination (NT1) at the user site via the ISDN U-interface and the line termination (LT) unit of the ISDN switch. This task completion report associates the ISTA hardware development directly with the design.

1.3 Document Overview

This document defines the ISTA proposed in the SCAR contract. The ISTA will be used to demonstrate communications of an ISDN terminal with another ISDN terminal through a satellite link. The satellite link could be furnished by either an ACTS or a Ku band earth station terminal provided that a 160 Kbps full-duplex bandwidth is allocated for the ISTA. All the ISDN services available to a user connected directly to a local exchange will be available to a remote ISDN user with the exception that the service will have a propagation delay of about 250 milliseconds.

This task completion report begins by describing the objectives of the ISIS hardware experiment in terms related to a communications satellite connected to an ISDN terrestrial link. A specific application of sending compressed video from NASA Lewis in Cleveland, Ohio to the GTE #5ESS switch in Chantilly, Virginia is postulated as a context for discussions for the development of the ISTA.

The ISTA development is decomposed into several detailed development views identifying the design refinements along the way. These development views are described in terms of their associated hardware, the chip set, and the software design. The ISDN basic access superframe structure and the satellite link access HLDC Frame Structure are described down to the bit level.

The ISTA activation and deactivation protocols for both the exchange and terminal equipment are presented in the context of end-to-end Z-diagrams. The relationship between the 68000 Development System and the ISTA is shown and a detail circuit diagram down to the chip pin connections is provided.

Two appendices provide the complete ISTA details. Appendix A lists the Bill of Materials and the presents the detail circuit diagrams for the ISTA hardware. Appendix B provides a software flow diagram for the software used in the ISTA hardware as well as a complete listing of the software.

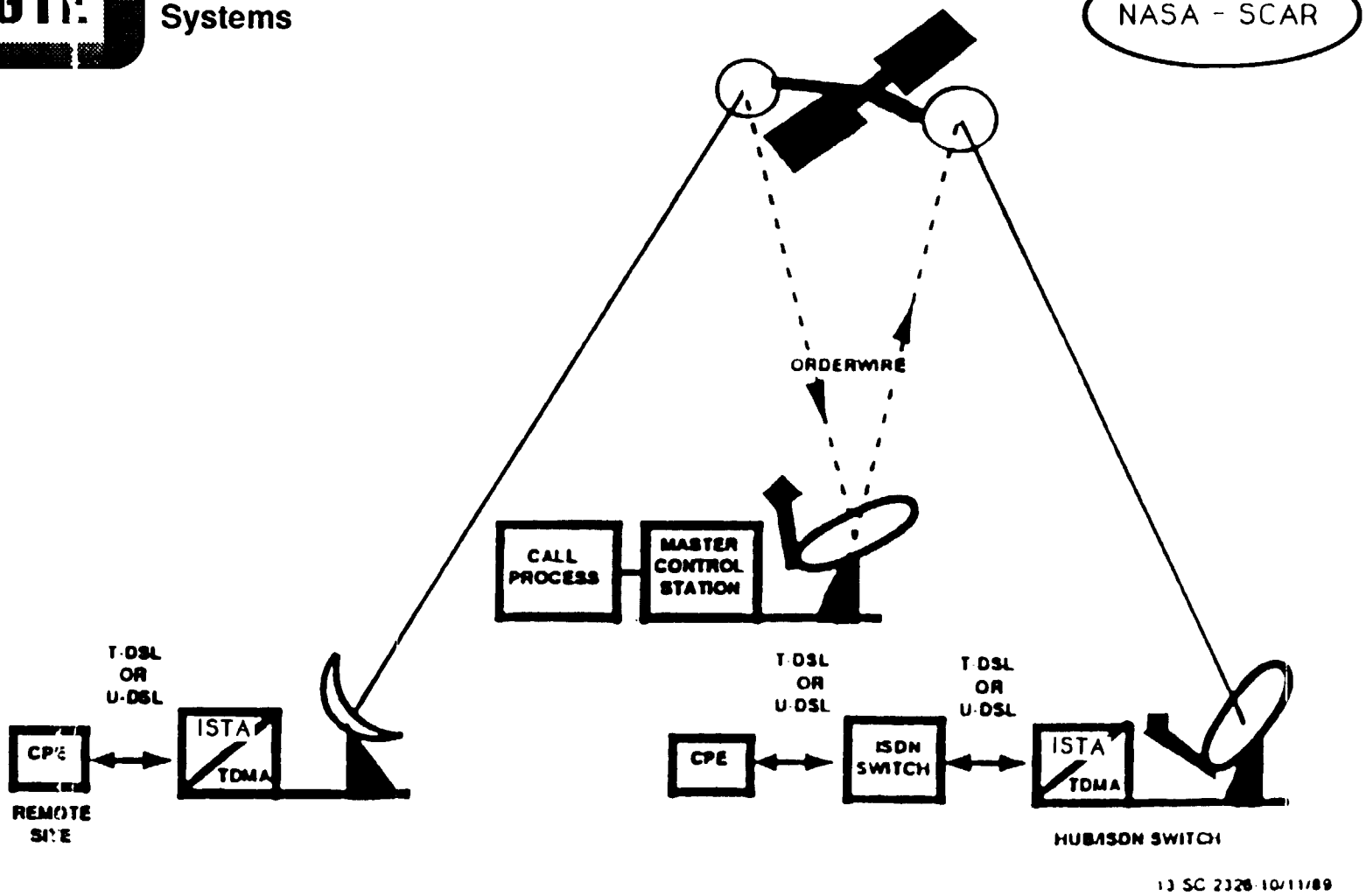


Figure 1.2-4 ISIS System Configuration for Remote Access

SECTION 2

POTENTIAL ISDN HARDWARE EXPERIMENTS

2.1 ISDN Hardware Experiment Objective

The objective of the ISDN Hardware Experiment is to demonstrate the feasibility of using typical communications satellites to connect ISDN users to ISDN exchanges via a non-ISDN Communications Satellite Link. Figure 2.1-1, "ISIS Hardware Development" shows the top view of a user terminal connected to a #5ESS Switch via line termination and network termination. The ISTA converts the ISDN Basic Access Superframe Structure into Satellite Link Access HDLC Frame Structure suitable for transmission via satellite. The ISTA design must also be capable of reversing the process on the network side of the satellite connection.

2.2 ISDN Typical Basic Access - Terrestrial/Satellite Links

Figure 2.2-1, "ISDN Typical Terrestrial/Satellite Links", shows customer premises connected to an ISDN switch at a local telephone exchange by a U-interface with 3.5 miles of twisted pair copper wire. This connection between the NT1 unit and the line termination (LT) provides the user with all the access for basic rate ISDN services.

Replacing this copper wire with a satellite link requires matching both the NT1 and the LT termination in terms of bit transfer, protocol timing and data rate adaption related to CCITT time-out values. Both the satellite and the corresponding ground system must be capable of supporting the typical ISDN 160Kbps basic access rate. The ISTA ensures that the protocol and user data conversions permit the timely support of the ISDN protocol and data.

2.3 Potential Application of ISTA

One of the postulated demonstrations of the ISTAs is to provide ISDN connectivity between the NASA Lewis Complex and the GTE #5ESS at Dulles International Airport. Figure 2.3-1, "Potential ISDN Satellite Connectivity", shows a compressed video image at NASA Lewis, Cleveland, Ohio being transmitted to GTE Chantilly, Virginia via ISTA equipped ground terminals. From GTE-Chantilly the ISDN frames pass through a U-interface in the Brite Channel Bank across 10 miles of fiber optic link to a Brite channel bank in the GTE Dulles #5ESS ISDN Switch. Tests of throughput and response-time can be made using this configuration or other similar configuration. The principal message for this report is that the ISTA provides the necessary conversion between the ISDN world and the communications satellite world.

2.4 Typical Application of ISTA

Figure 2.4-1, "Typical Remote Site ISDN Application with ISTA" shows a more detail view of the application of these ISTA devices. The twisted pair connection to the NT1 or the ISDN switch provide generic access to typical teleco services. Whereas, the RS-449 connectivity to satellite modems permit the use of communications satellites as commmunication components of any ISDN network.

GTE Laboratories Incorporated

NASA - SCAR

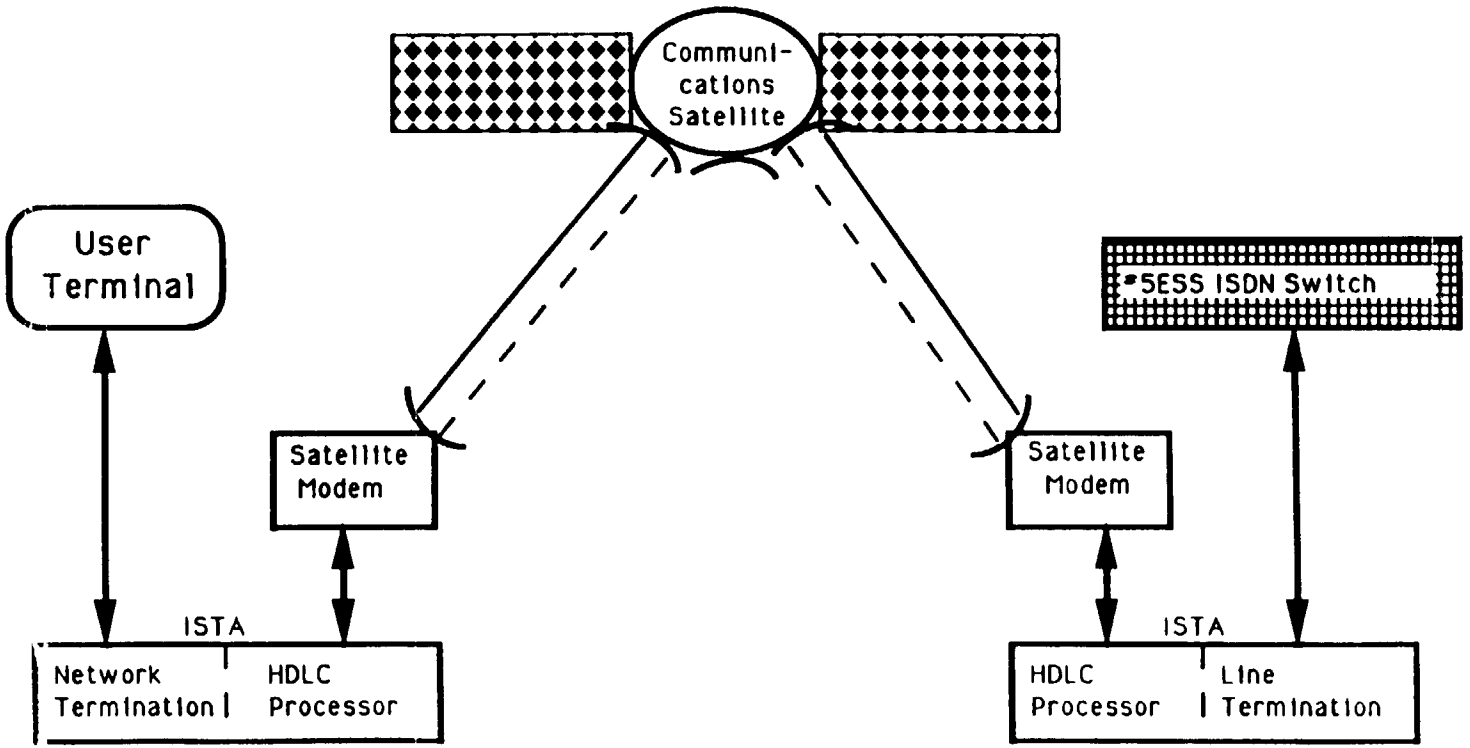


Figure 2.1-1 ISIS Hardware Development

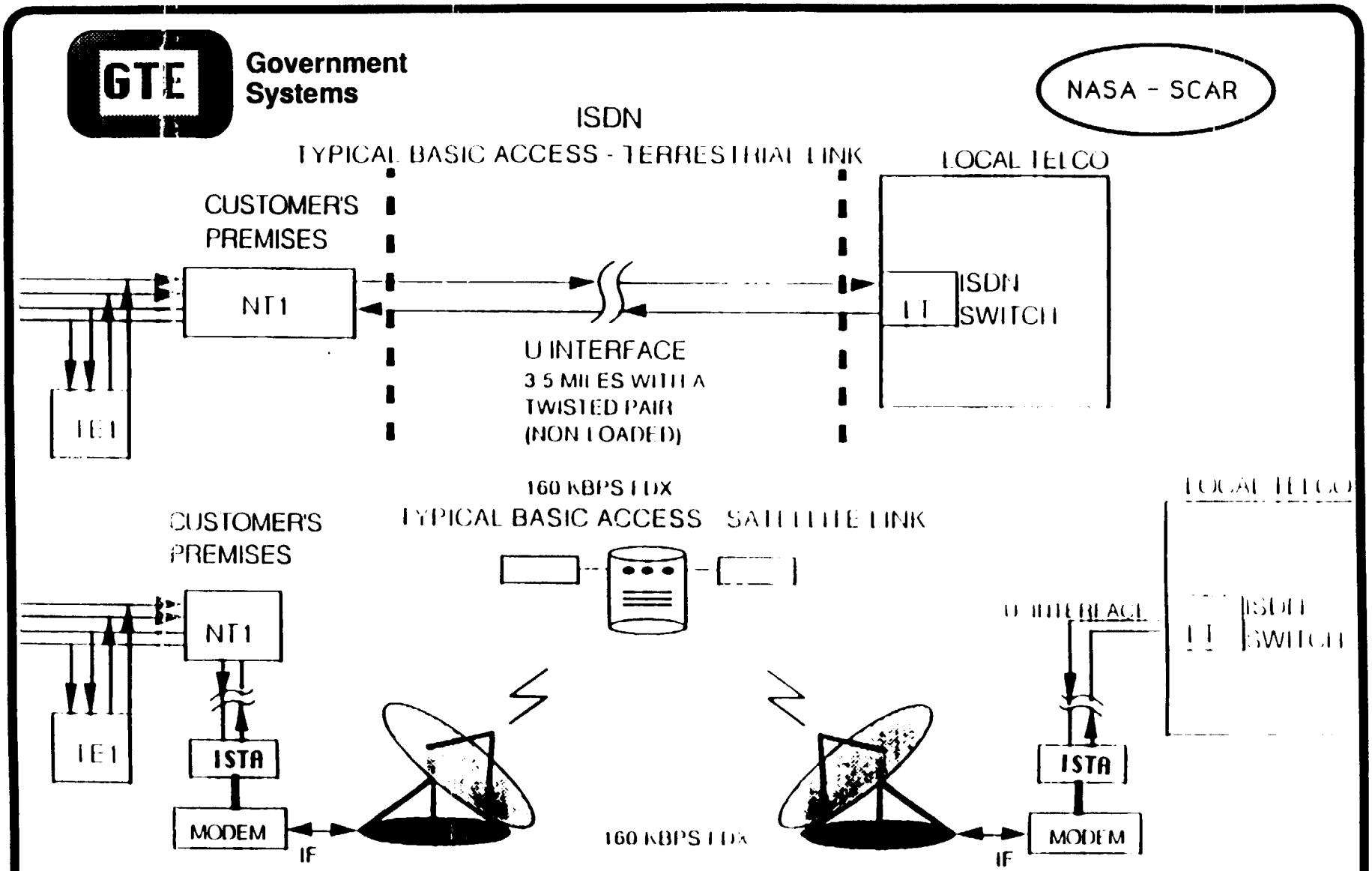


Figure 2.2-1 ISDN Typical Terrestrial/Satellite Links

GTE Government Systems

NASA - SCAR

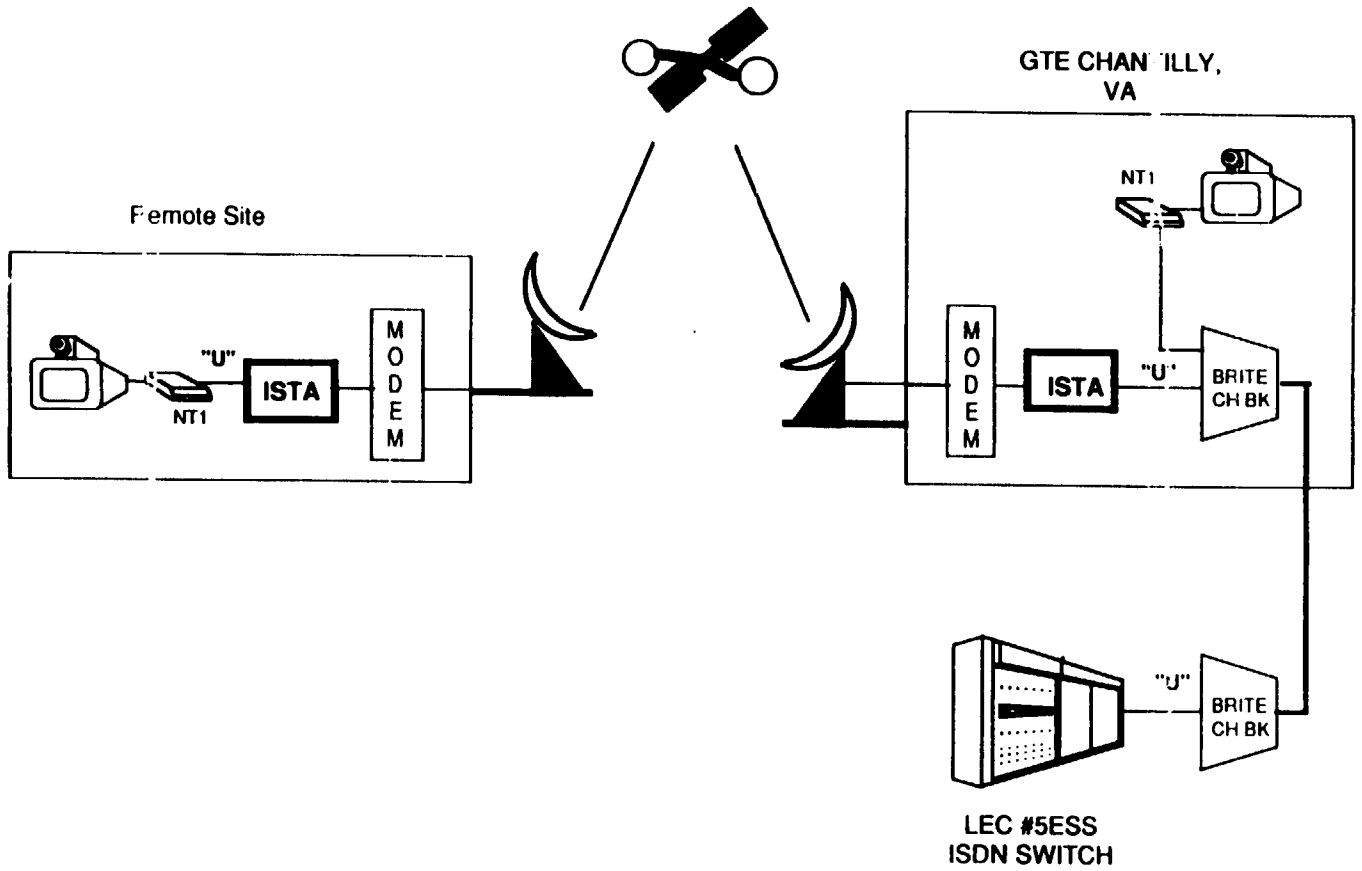
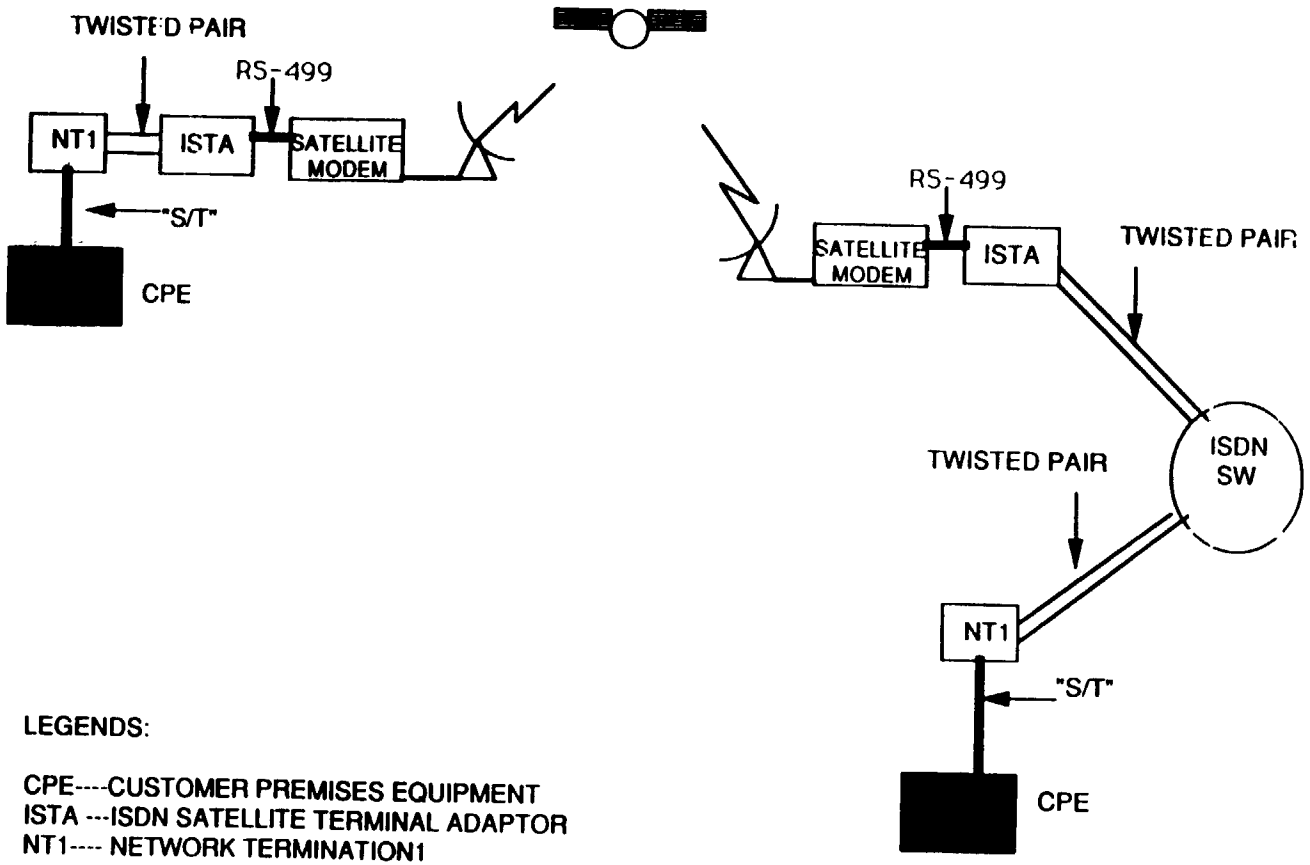


Figure 2.3-1 Potential ISDN Satellite Connectivity

8 T038 SCA DAT
Pot ISDN Satel Connectivity
April 20, 1992



LEGENDS:

CPE----CUSTOMER PREMISES EQUIPMENT
ISTA ---ISDN SATELLITE TERMINAL ADAPTOR
NT1---- NETWORK TERMINATION1

Figure 2.4-1 Typical Remote Site ISDN Application with ISTA

8 T058 SCA DAT
Typical Remote Connectivity
April 20, 1992

SECTION 3

ISTA HARDWARE DEVELOPMENT

3.1 ISTA Top View

At the top level, the ISTA interfaces the U-interface with the RS-499 interface at the 160Kbps rate. Figure 3.1-1, "ISDN Satellite Terminal Adapter (ISTA)", shows the ISTA between the user terminal and the Low Bit Rate Terminal (LBR-2). The expanded view at the LT and HDLC level, and the development using the MC145472, the MC68302, and RS-449 Driver/Receiver chip set are discussed in subsequent sections.

3.2 ISTA Functional Block Diagram

Figure 3.2-1, "ISTA Functional Block Diagram", shows both the CPE side and the switch side of the ISTA. For the CPE side the U-interface connects the user NT1 to a line terminal that is connected to a HDLC processor that converts the basic access frames to the RS-449 frames for the communications satellite. On the switch side of the ISTA the RS-449 frames are converted by the HDLC processor to provide ISDN basic access frames between the NT unit and the LT unit of the #5ESS ISDN Switch. Figure 3.2-2, "Specific Network and Line Terminations", depict the ISTA device with its development parameters.

3.3 ISTA Frame Structures

Each side of the ISTA has its unique frame structure to accommodate their respective protocols. Figure 3.3-1, "ISTA Frame Structures", shows both frame structures. The ISDN Basic Access Superframe Structure uses a format of 1920 bits. The transmission across the U-interface is organized into groups of eight 2B1Q frames, called superframes. A frame consists of three fields:

Synchronization word (SW): Used for physical layer synchronization and frame alignment. It consist of a pattern of 18 bits.

User Data (12(2B+D): 12 groups of 2B and D information. Each group contains 8 B1 bits, 8 B2 bits, and 2 D bits resulting in 216 bits of user data.

Overhead Data: These bits are used for physical channel maintenance, error detection and power status. A total of 6 bits are used per frame.

As shown in Fig 3.3-1 the inverted synchronization word (ISW) identifies the first frame in the superframe; its is a pattern of 18 bits that is merely the inverse of the normal synchronization word. The superframe organizes the 6 overhead bits of each frame into a block of 48 bits

The satellite link access HDLC frame structure consists of 1808 bits apportioned in a different manner. The eight frames of user information are combined into a single frame of 1728 bits for the 96(2B+D). The overhead bits are collected into Flag, Control, CRC, M-bits and Fill. The fill is used to perform rate adaption between the terrestrial and satellite protocols.

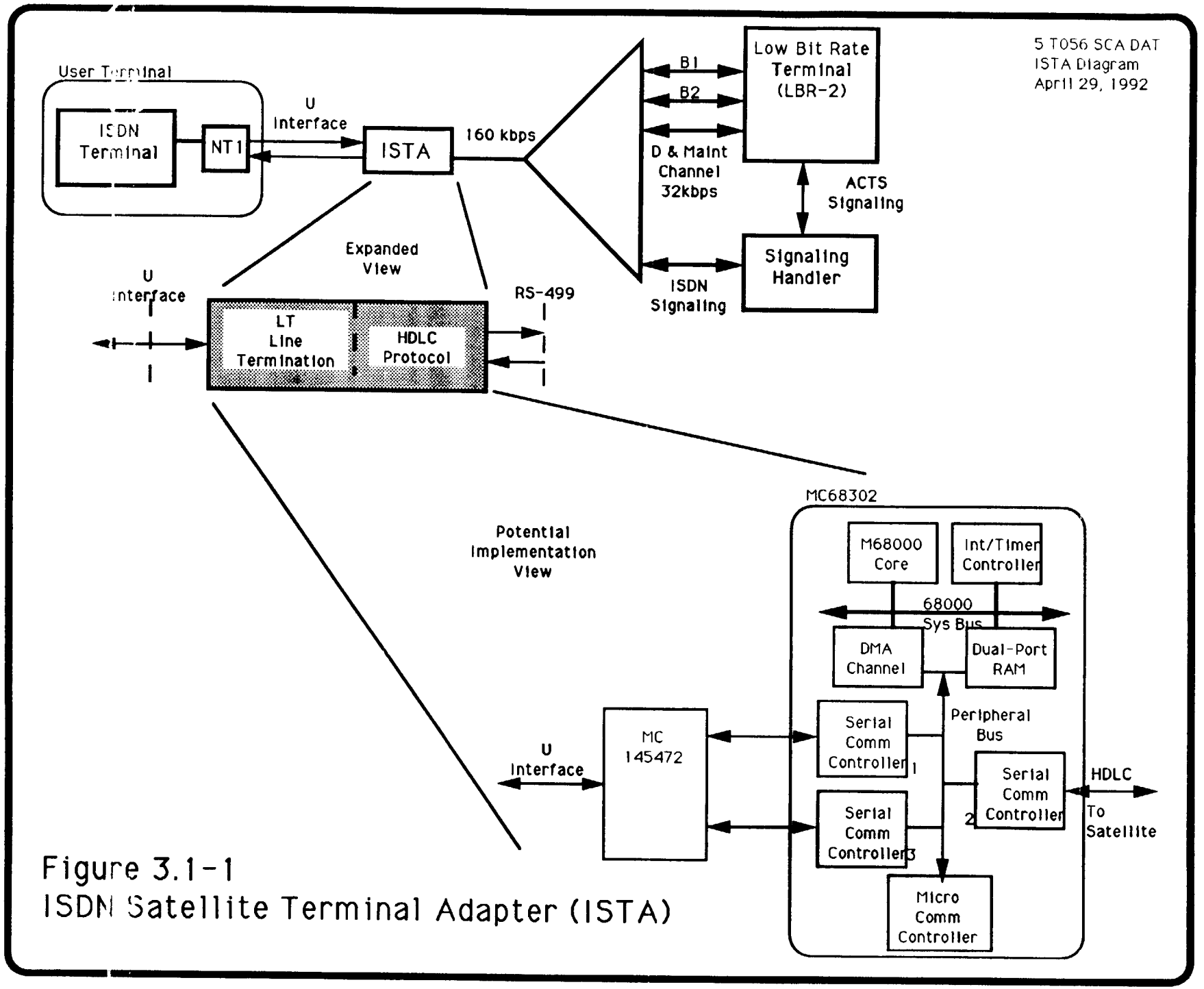


Figure 3.1-1
ISDN Satellite Terminal Adapter (ISTA)

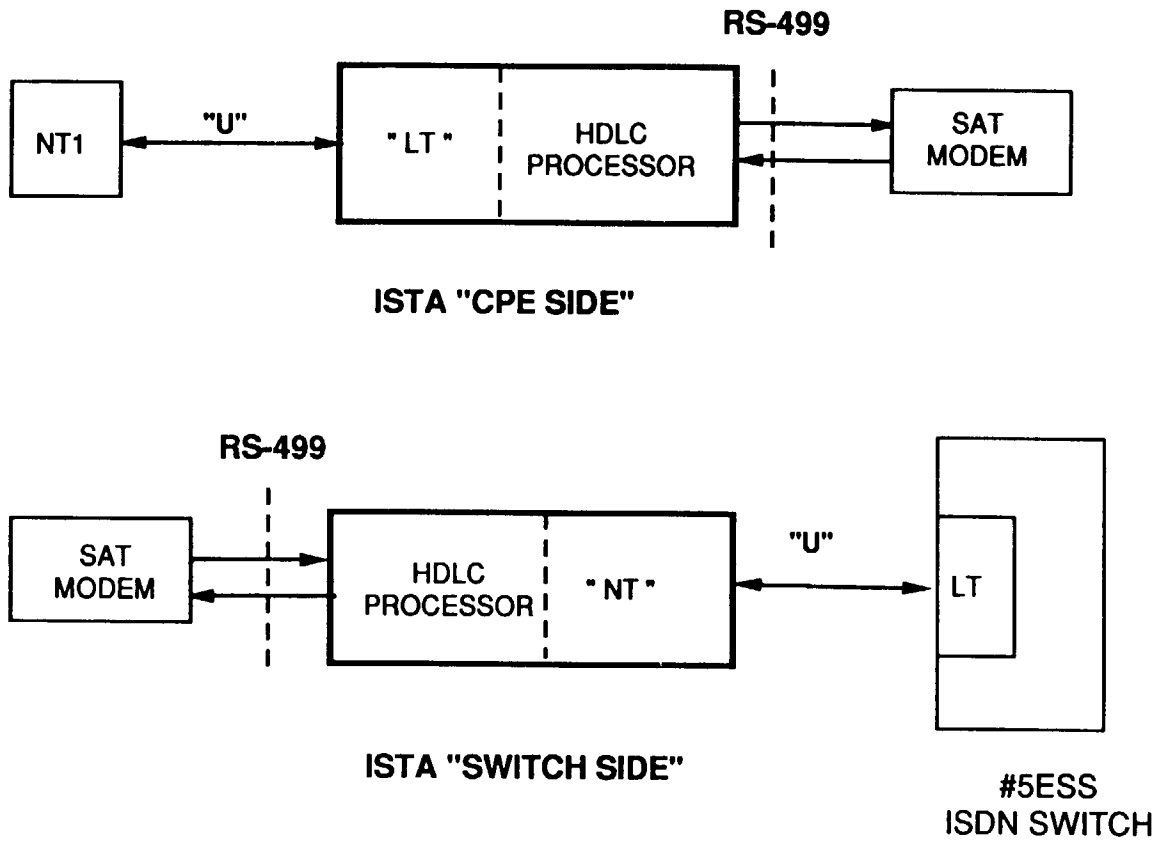
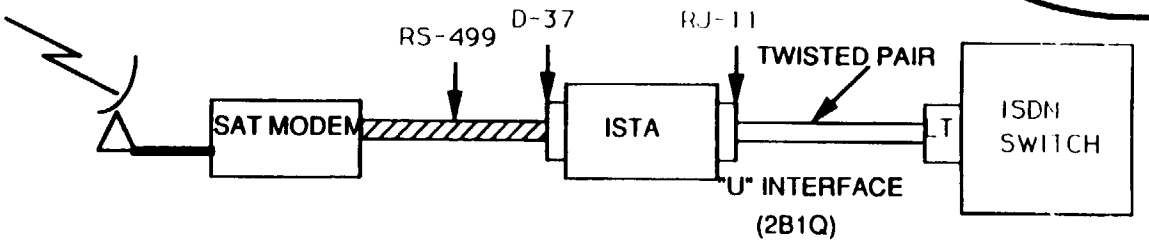
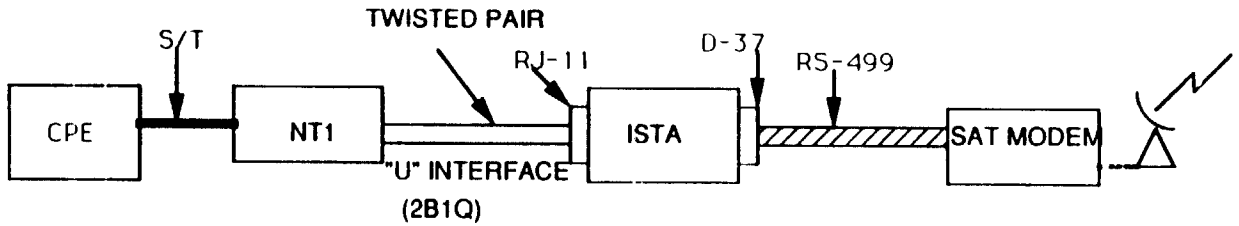


Figure 3.2-1 ISTA Functional Block Diagram



NETWORK TERMINATION

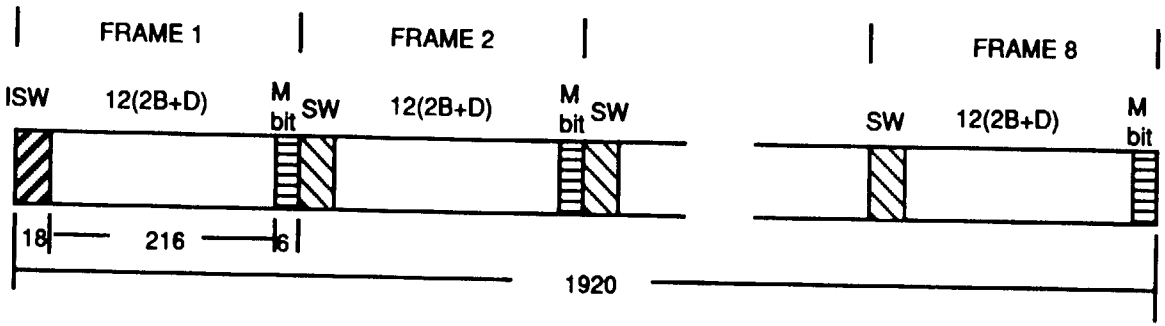


LINE TERMINATION

LEGENDS:

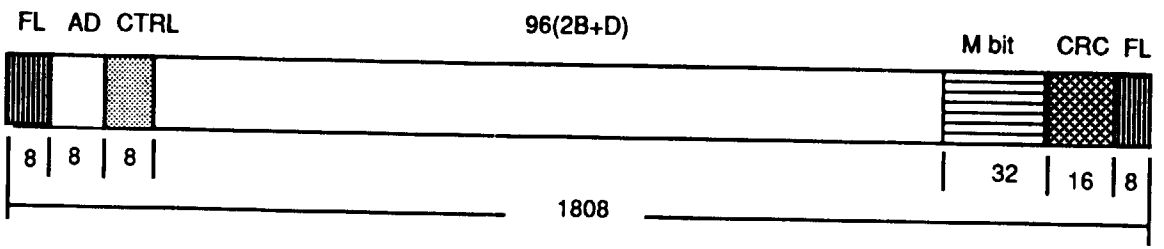
CPE---- CUSTOMER PREMISES EQUIPMENT
NT1----- NETWORK TERMINATION 1
LT------- LINE TERMINATION
SAT MODEM---SATELLITE MODEM

Figure 3.2-2 Specific Network and Line Terminations



ISDN Basic Access Superframe Structure

B = 8 bits
D = 2 bits
Data Rate = 160 Kbps



Satellite Link Access HDLC Frame Structure

Figure 3.3-1 ISTA Frame Structures

3.4 ISTA Chip Set

The ISTA design shown in Figure 3.4-1, "ISTA Hardware Block Diagram", includes using the MC145472 ISDN U-Interface Transceiver, the MC68302 Multi-Protocol Processor and added RAM/ROM for suitable memory. The serial communication controllers on the MC68302 are connected to RS-449 drivers and receivers used to drive the ISDB U-interface transceiver. The third serial communications controller connected to the same MC68302 peripheral bus as the other two communications controllers provides HDLC frames to the RS-449 line Tx/Rx function.

This same ISTA design is capable of supporting the CPE side and the switch side of the interface. To synchronize with satellite timing the U145472 derives timing from the received satellite clock pulses using in a phase lock loop to control the ISDN U-interface transceiver when the ISTA is used on the switch side - Switch to Satellite interface. The same loop timing switch is open when the ISTA is used on the CPE side - NT1 to Satellite interface. Figure 3.4-2, "Network Timing Diagram", shows the signal interactions among the chips used in the development of the ISTA hardware.

3.5 ISTA Software

The ISTA design uses off-the-shelf chip sets that require principally pin to pin circuit connectivity. These chips, however, rely on digital instructions to perform their transmission, reception, and protocol frame conversion processes. Figure 3.5-1, "ISTA Software Flow Diagram", depicts the top level flow diagram for the ISTA software. After the sequential initialization of the MC68302, the HDLC Comm, the 2B+D Comm and the SCP the software selects the U interface initialization depending on the ISTA switch setting. After all these initialization on both ISTAs the respective software starts the appropriate activation procedure and waits for an interrupt.

These interrupt service routines include:

* M4 Bits Processing
* Activation/Deactivation
* Embedded Operation Channel Processing
* IDL "2B+D" Tx and Rx Buffer Processing
* HDLC Tx and Rx Buffer Processing

The ISTA software is being developed on a system like the 68000 Development System using the MC145494 Evaluation Kit and ADS302 Development System shown in Figure 3.5-2, "ISDN Satellite Terminal Adapter Development System". A complete listing of the source code for the software used in the ISTA is presented in Appendix B.

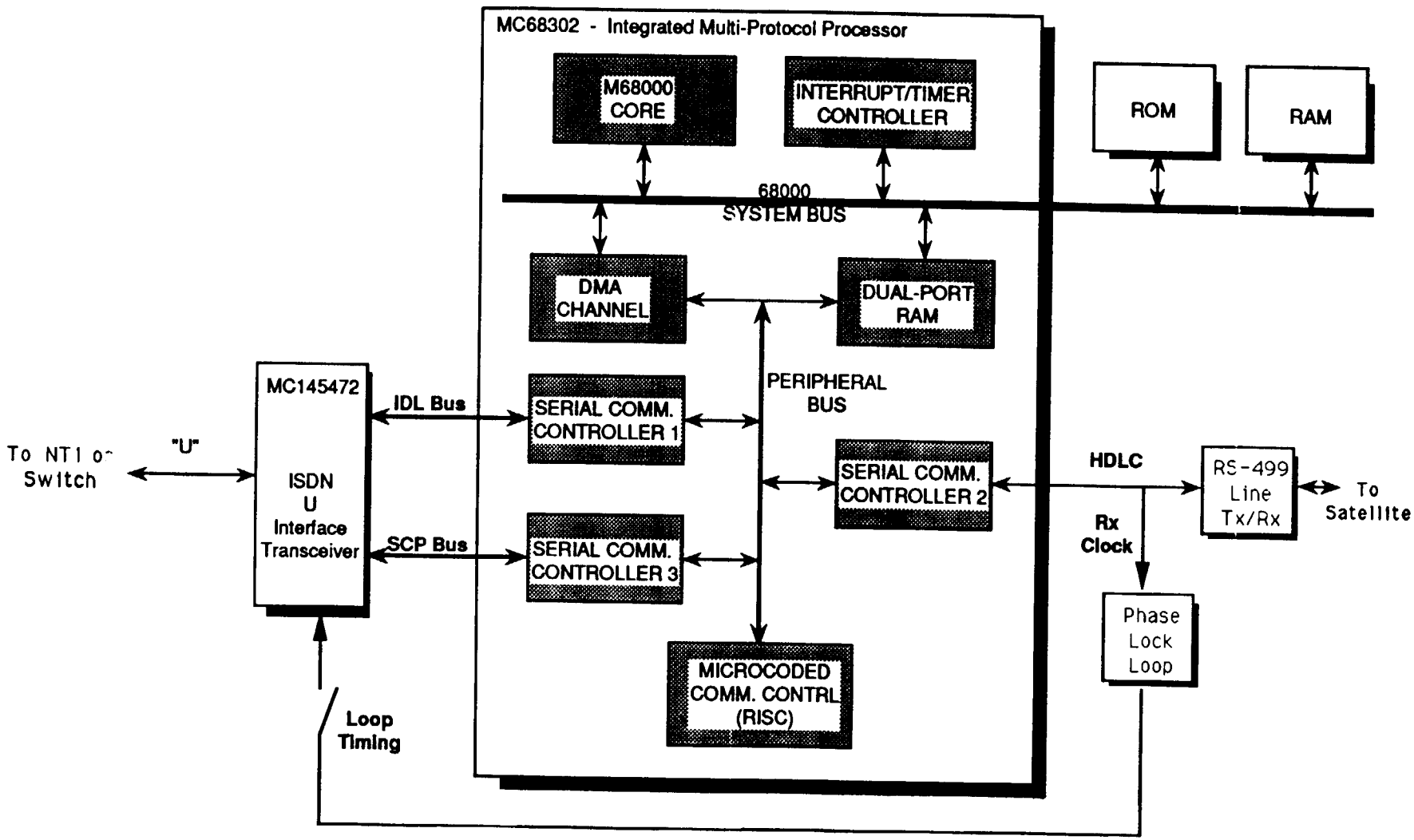


Figure 3.4-1 ISTA Hardware Block Diagram

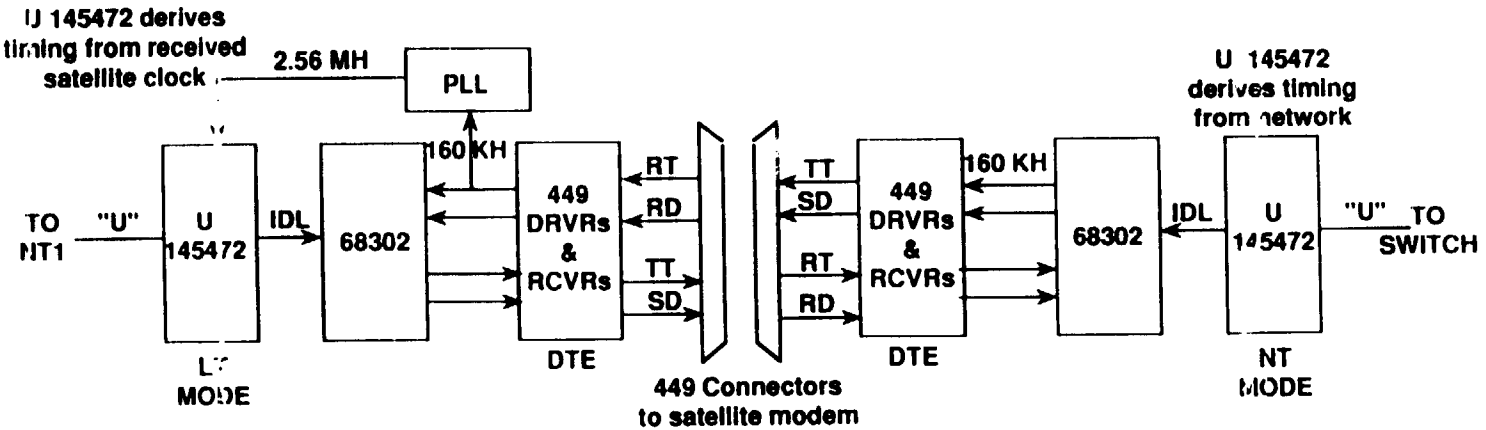


Figure 3.4-2 Network Timing Diagram

8 T058 SCA DAT
Typical Remote Connectivity
April 20, 1992

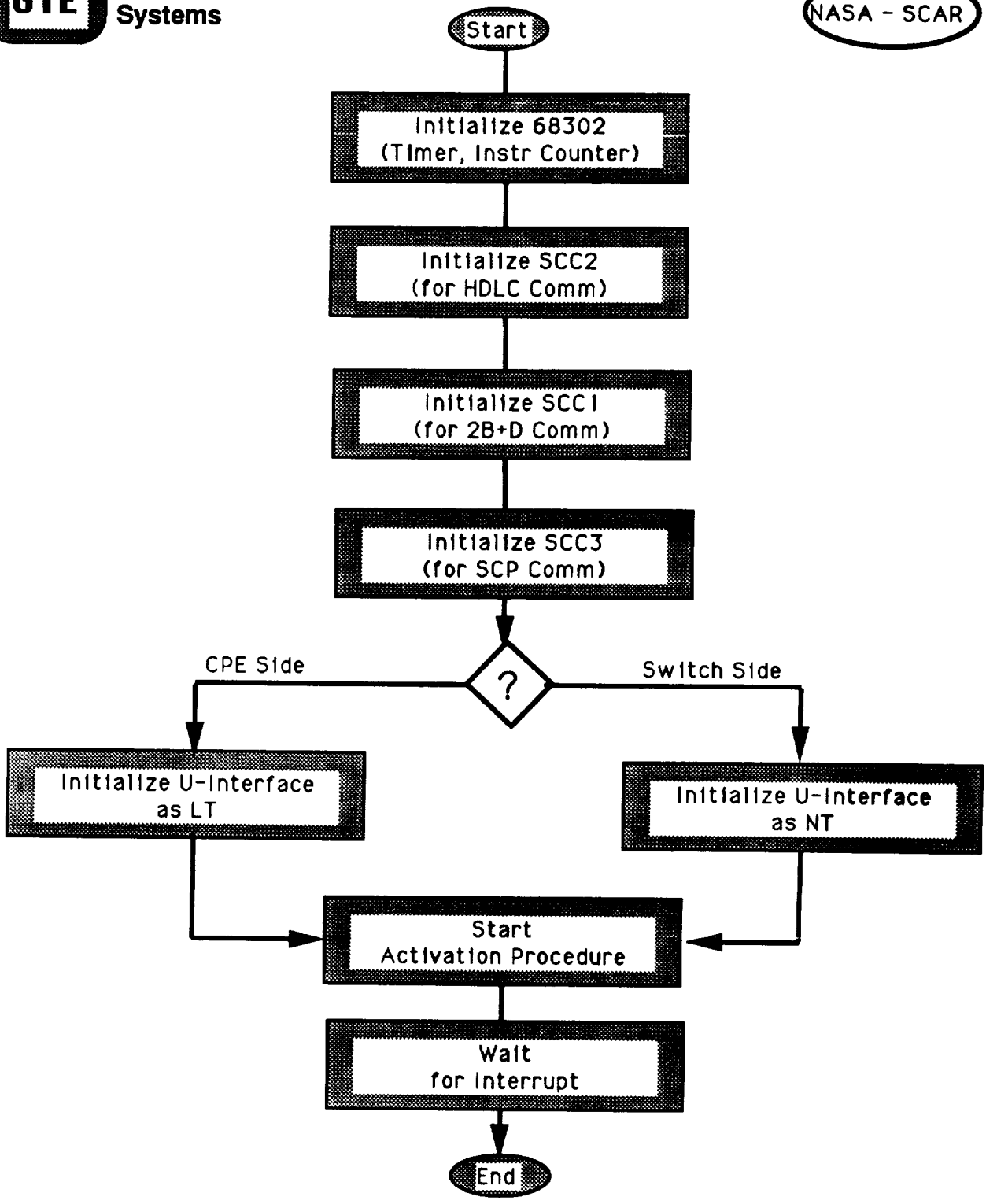


Figure 3.5-1 ISTA Software Flow Diagram

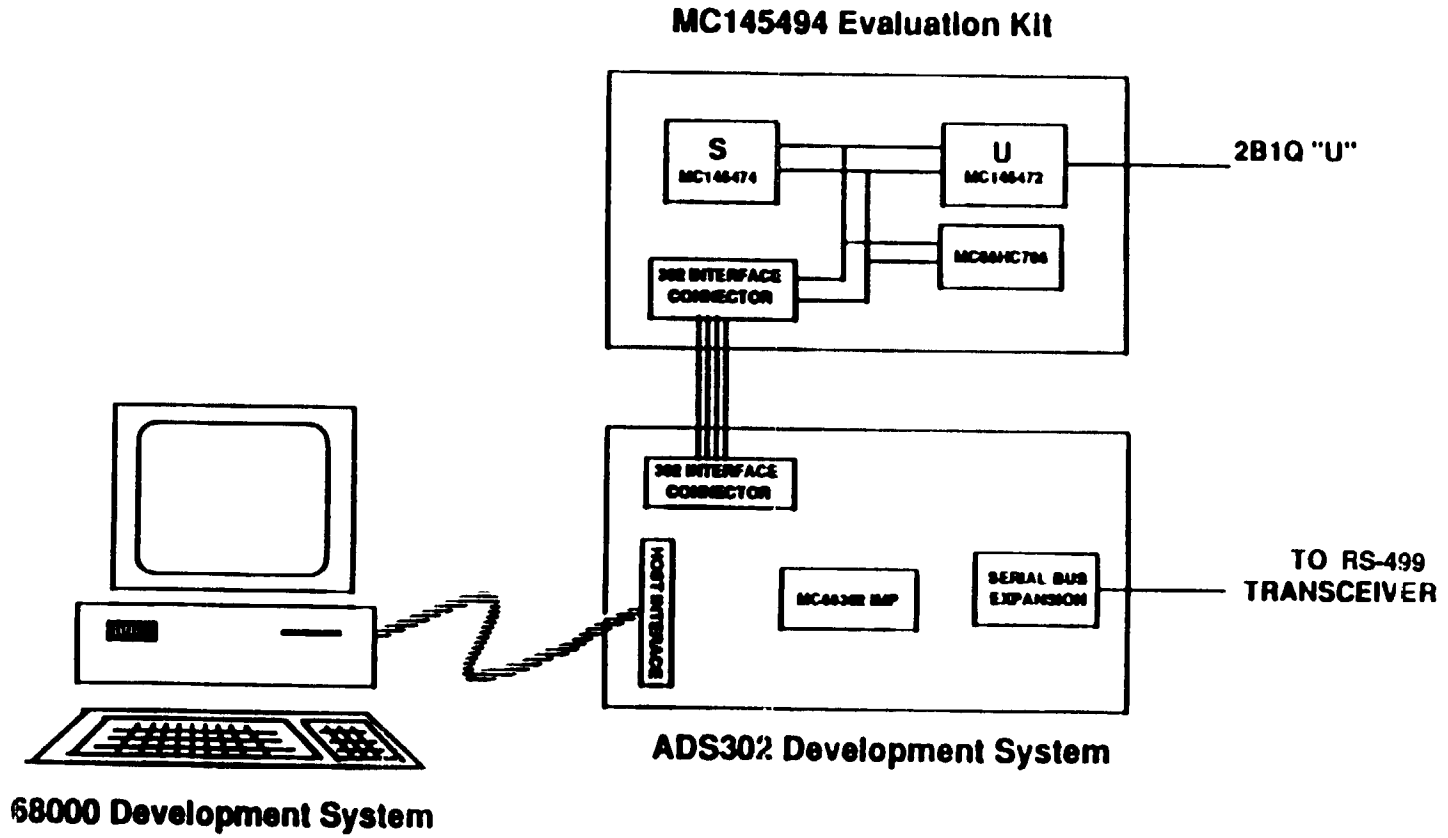


Figure 3.5-2 ISDN Satellite Terminal Adapter Development System

SECTION 4

ISTA Activation Diagrams

4.1 Introduction

Before meaningful ISDN communication services can be provided through the satellite, each ISTA must be activated with the proper protocol sequence in conjunction with its counter part ISTA at the other end. In that context, activations can be viewed as being initiated by either the network exchange or initiated by the terminal equipment.

4.2 ISTA Activation Initiated by the Exchange

Figure 4.2-1, "Total Activation Initiated by the Exchange", shows the sequence of protocols initiated by the exchange and the responses that will permit the ultimate communication of ISDN user traffic. The Z-chart of these protocol sequences is aligned with a top level block diagram of the satellite connectivity between the ISDN switch and the terminal equipment (TE). The exchange begins the activation process from its null state when it receives an activation request (AR) protocol message. The switch LT changes its state to activation proceeding (AP) and sends an activation request to its local ISTA, NT>ISTA, which immediately sets its state to the activation proceeding (AP) state. The AR protocol message received by the NT section of the ISTA converts the digital AR message into control bits in the HLDC frames that are continually being sent through the communications satellite to the other ISTA, SAT>LT, and sets its state to AP. The NT>SAT ISTA begins the exchange of synchronizing information with the LT at the Exchange.

Meanwhile the HDLC protocols with the set control bits are received by the SAT>LT ISTA which sets its state to AP and begin exchanging synchronizing information with the user terminal's NT. That NT also sets its state to AP. The superframe synchronization (SS) on the each side of the satellite results in both the exchange-LT and the equipment -NT being set to the SS along with their corresponding ISTAs. The user-NT sends an Info2 message indicating it is ready to receive user data as Info3. The proper reception of Info3 sets the states of all the interfaces to activated (act=1) along the way and an activation indication (AI) state is entered. The timely response to an Info3 message by an Info4 message keeps all the interfaces synchronized for continuous information flow.

4.3 ISTA Activation Initiated by the Terminal Equipment

Figure 4.3-1, "Total Activation Initiated by Terminal Equipment", shows the sequence of protocols initiated by the exchange and the responses that will permit the ultimate communication of ISDN user traffic. The Z-chart of these protocol sequences is aligned with a top level block diagram of the satellite connectivity between the ISDN switch and the terminal equipment (TE). The terminal equipment begins the activation process from its null state by receiving an activation request (AR) protocol message in the form of Info1 from the terminal equipment. The protocol exchange process continues in the same fashion as described above until all interface states are in the activated state (act=1), signaling activation indication (AI), and a superframe synchronized flow of information is flowing in terms of Info3 and Info4.

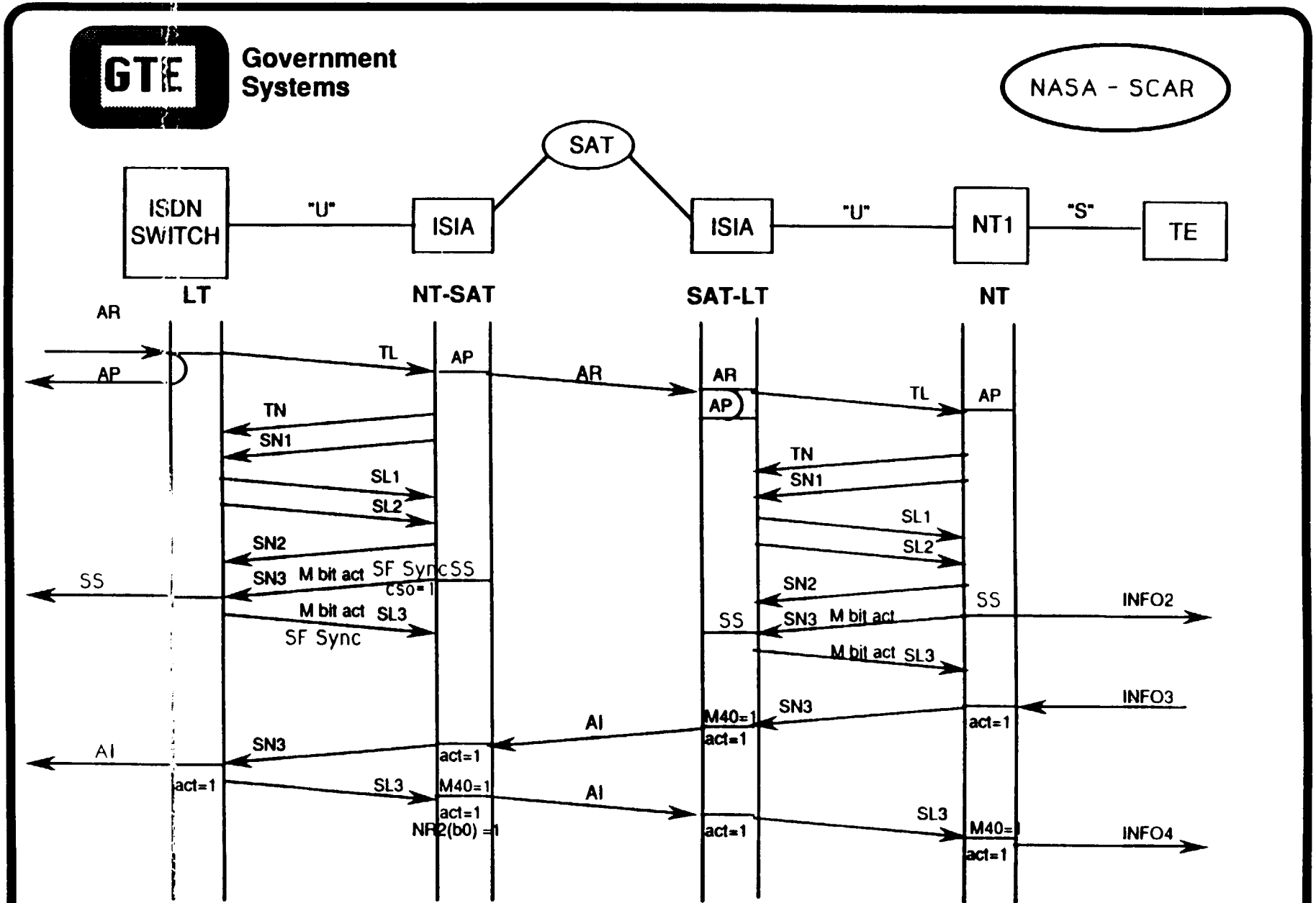


Figure 4.2-1 Total Activation Initiated by the Exchange

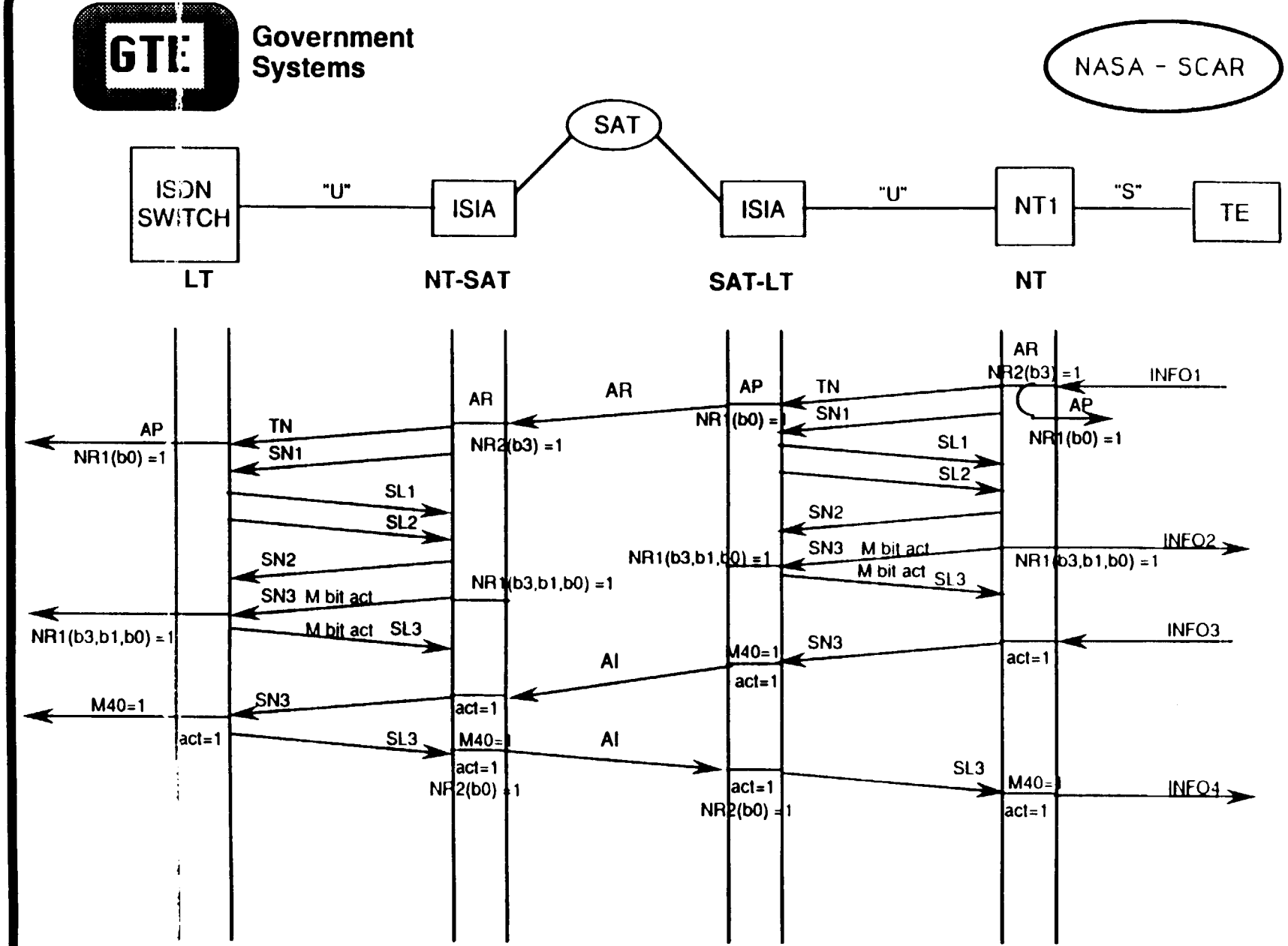


Figure 4.3-1 Total Activation Initiated by Terminal Equipment

4.4 ISTA Deactivation Process

Figure 4.4-1, "Total Deactivation Process", shows the sequence of protocols initiated by the exchange and the responses that will permit the deactivation of all the interfaces. The Z-chart of these protocol sequences is aligned with a top level block diagram of the satellite connectivity between the ISDN switch and the terminal equipment (TE). The LT on the exchange side receives a deactivation request (DR) that is passed along in term of state changes, protocol messages, and reset at the interfaces Then at the terminal user end the NT1 and TE continue to exchange Info0 protocol messages.

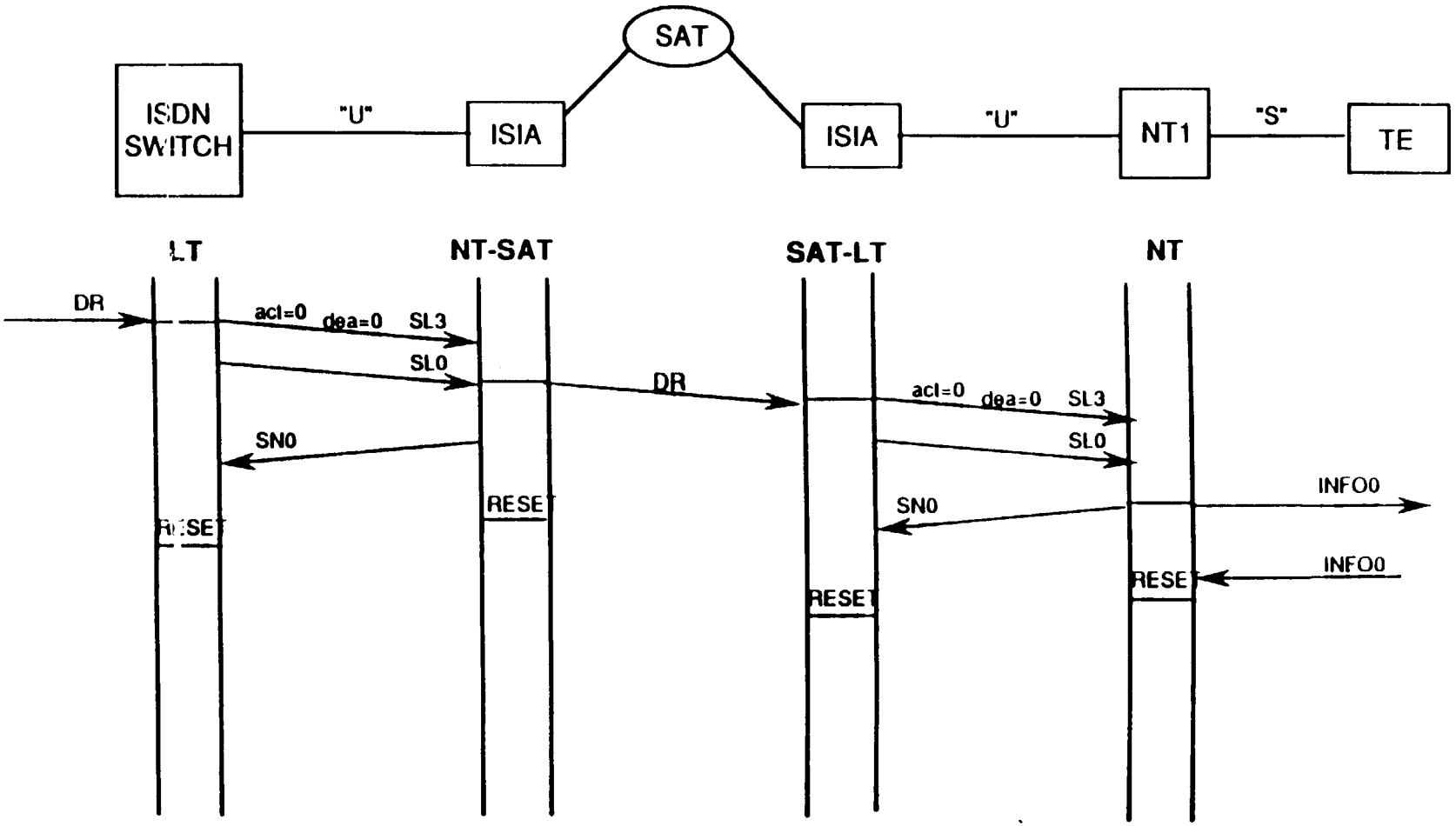


Figure 4.4-1 Total Deactivation Process

SECTION 5

SUMMARY

5.1 General

This task completion report for the ISIS Hardware experiment development presented the complete end-to-end view from the basic objectives to the ISTA circuit design. The ISTA hardware development is applicable to the the Interim Service ISDN Satellite (ISIS) and can be used with any HDLC interfaced communications satellite. The ultimate aim of this aspect of the SCAR Program is to demonstrate that ISDN communications via satellite is possible and to corroborate the engineering design values for new advanced ISDN communications satellite. The technical and operational parameters for this ISDN advanced communications satellite design will be obtained from an engineering software model of the major subsystems of the ISDN communications satellite architecture. Discrete event simulation experiments will be performed with these ISIS models using various traffic scenarios, technical parameters, and operational procedures. The data from those simulations will be analyzed using the performance measures discussed in previous NASA SCAR reports. These same data will be compared from data gathered from hardware experiments using ISTAs cited in this report.

5.2 Review

This task completion report began by describing the objectives of the ISIS hardware experiment in terms related a communications satellite connected to an ISDN terrestrial link. A specific application of sending compressed video from NASA Lewis in Ohio to the GTE #5ESS switch in Virginia was postulated for discussions about the design of the ISTA.

The ISTA development was decomposed into several detailed views identifying the development refinements along the way. Each of these development views was described in terms of their associated hardware, the chip set, and the software development. The ISDN basic access superframe structure and the satellite link access HLDC Frame Structure were described down to the bit level.

The ISTA activation and deactivation for both the exchange and terminal equipment was discussed in the context of end-to-end Z-diagrams. The relationship between the 68000 Development System and the ISTA was shown and a detail circuit diagram down to the chip pin connections was provided together with a listing of the ISTA source code.

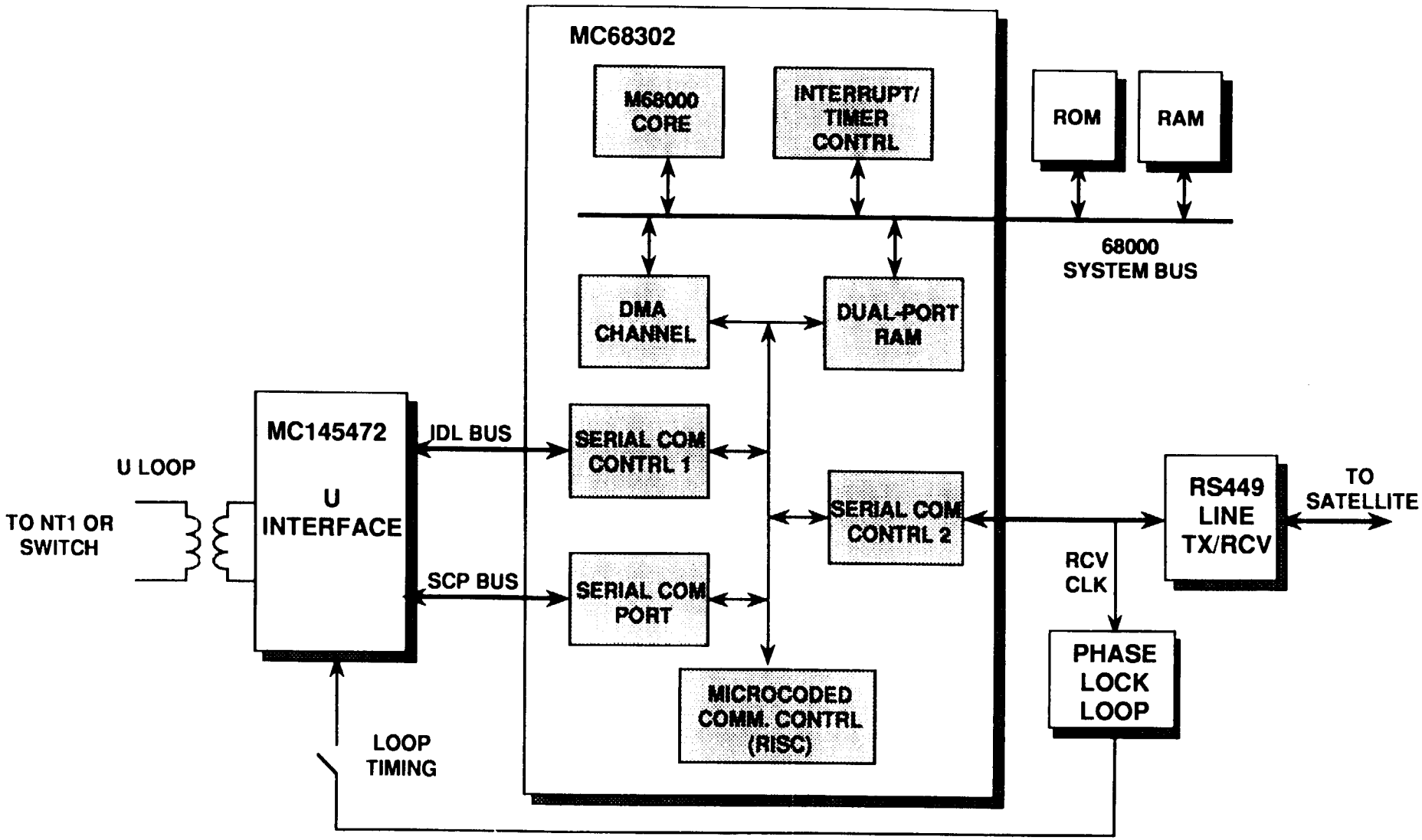
5.3 Continuing Efforts

The ISTA is NOT complete. The subcontractor is in the process pf building three ISTAs suitable for experiments with ACTS or other communications satellites. Experiments will be conducted to verify the performance of the ISTA and to demonstrate the proof of concept. The experiment configuration will include ISDN terminals, ISTAs, the ISDN hub switch and, optionally , the ACTS/TDMA simulator. An ISDN protocol emulator/analyzer will be used to verify protocol conformance of the ISTA interfaces. The ACTS/TDMA hardware simulator could be used to augment ISTA testing. The ACTS/TDMA simulator would provide a hardware simulation of ACTS call processing procedure, processing delays, and switching delays and would help the team to undertake more advanced ISDN communications tests. Experiments are expected to include ISDN call applications such as voice call management, video conference, and Group IV facsimile transmission in addition to various passive bus configurations. An update to this report will be provided after the subcontractor delivers the three ISTA's in July 1992.

APPENDIX A

ISTA HARDWARE CIRCUIT DIAGRAMS

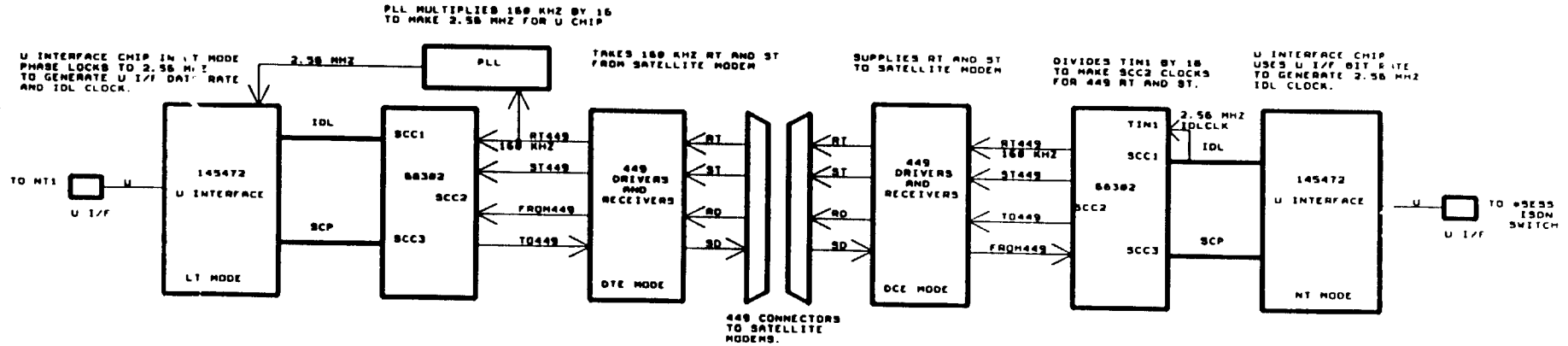
This appendix presents the circuit diagrams associated with the ISDN Satellite Terminal Adapter (ISTA). A top view of the ISTA is shown in the "ISTA HARDWARE BLOCK DIAGRAM" followed by a "CLOCK PASSING DIAGRAM" indicating the signal interactions among the chips. The next few charts show the pin by pin connections of these chips and other components. The last page of this appendix provides the "Bill of Material" for the ISTA.



MC145472 - ISDN U-Interface Transceiver

MC68302 - Integrated Multi-Protocol Processor

ISIA HARDWARE BLOCK DIAGRAM



NOTES:

1. U INTERFACE CHIPS SHOULD BE CONFIGURED FOR 16 BIT IDL MODE USING 2.56 MHZ CLOCKS. THE HARDWARE IS WIRED TO MAKE THE U I/F CHIPS IDL MASTERS. AS IDL MASTERS THEY SUPPLY THE IDL CLOCK AND SYNC.

2. FOR NT MODE CONFIGURE SCC2 RCLK AND TCLK AS OUTPUTS. FOR LT MODE CONFIGURE THEM AS INPUTS.

CLOCK PASSING DIAGRAM

UIFBD.SCH

KLT TELECOM INCORPORATED
7238 HIKMAT ROAD
MANASSAS, VA 22111

Title: U INTERFACE

Size: B | Document Number: | REV: A

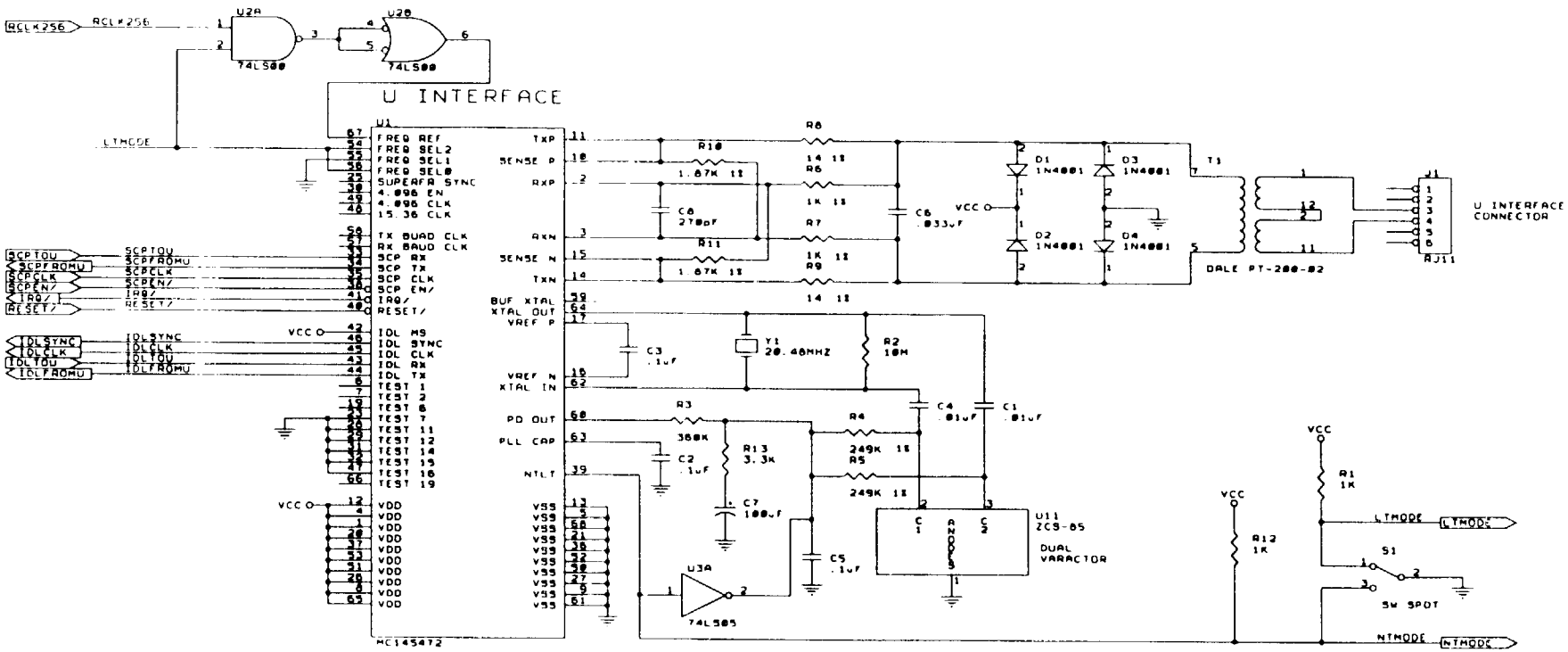
Date: Feb 07, 1995 | Sheet 1 of

!LINK

!UIF\UIF.SCH
!UIF\CPU.SCH
!UIF\MEM.SCH
!UIF\449.SCH

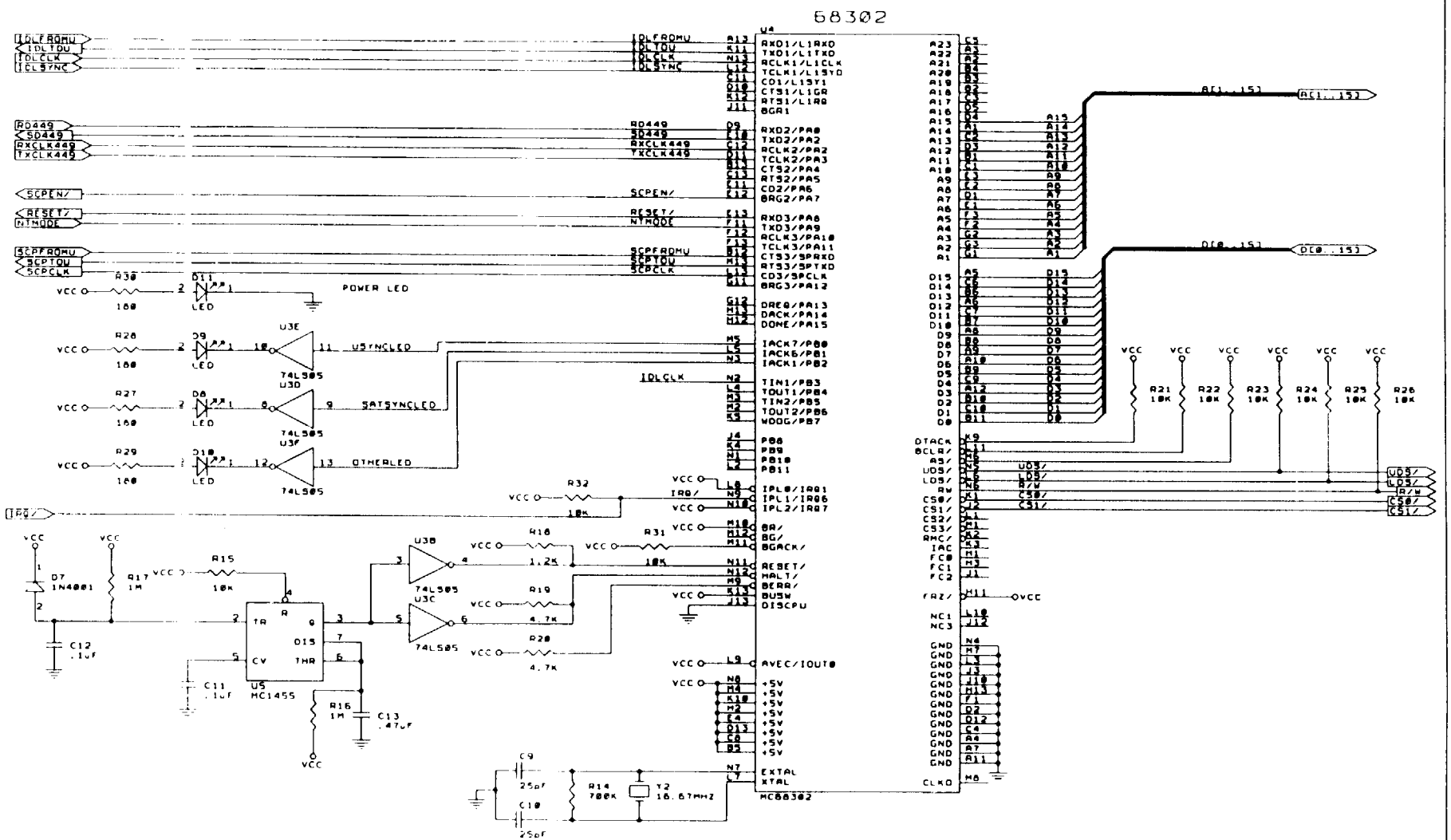
UIF.SCH

	KLT TELECOM INCORPORATED 7238 HIXMAT ROAD MANASSAS, VA 22111			
	U INTERFACE			
	SIZE B	FSCM NO	DWG NO	REV A
March 15, 1992	SCALE		SHEET 1 Of 4	



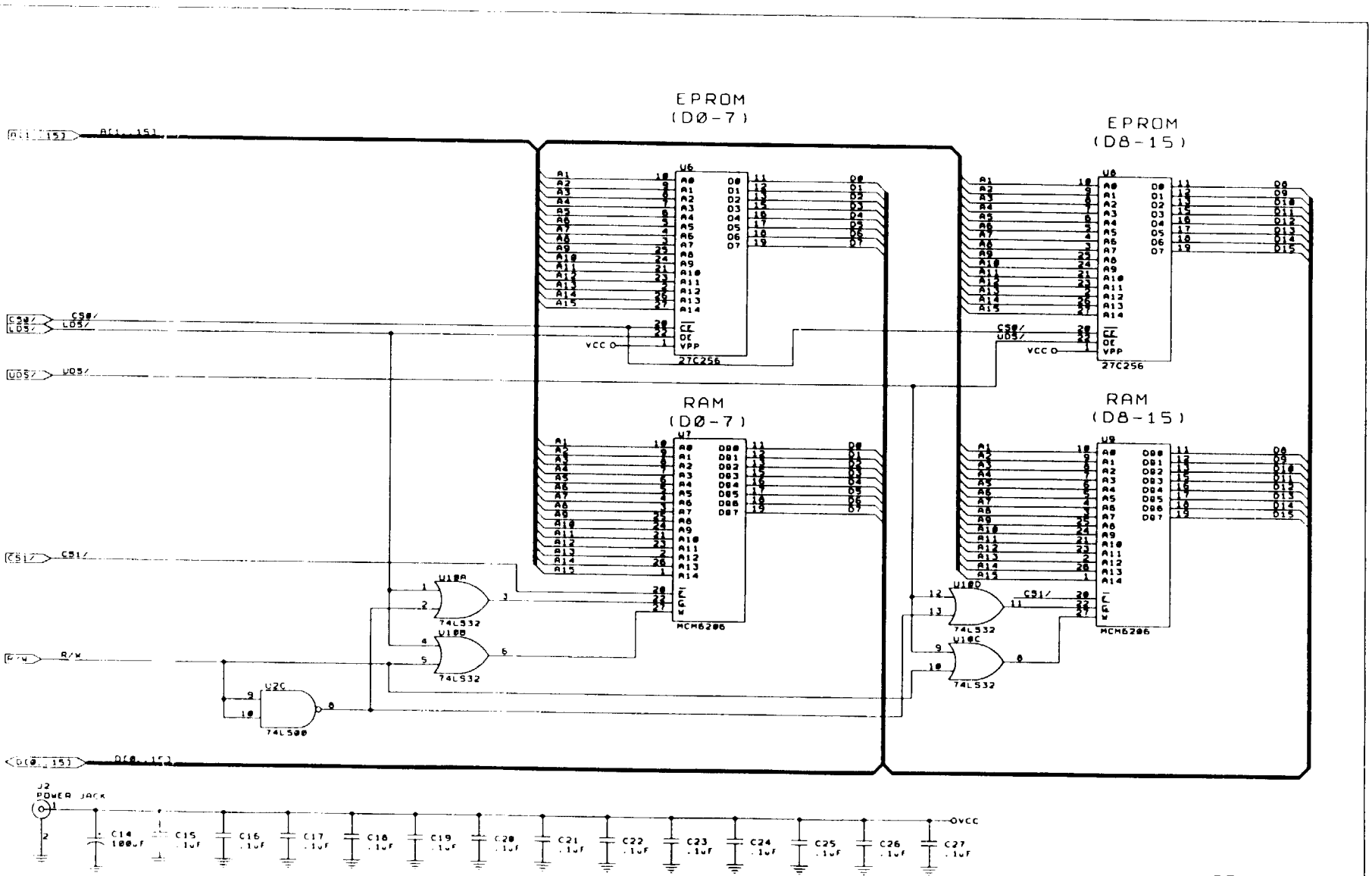
UIFUIF.SCH

	KLT TELECOM INCORPORATED 7238 MIKMAT ROAD MANASSAS, VA 22111			
	U INTERFACE			
	SIZE: B	FSCM NO	DWG NO	REV: A
March 23, 1992	SCALE		SHEET: 1 OF 4	



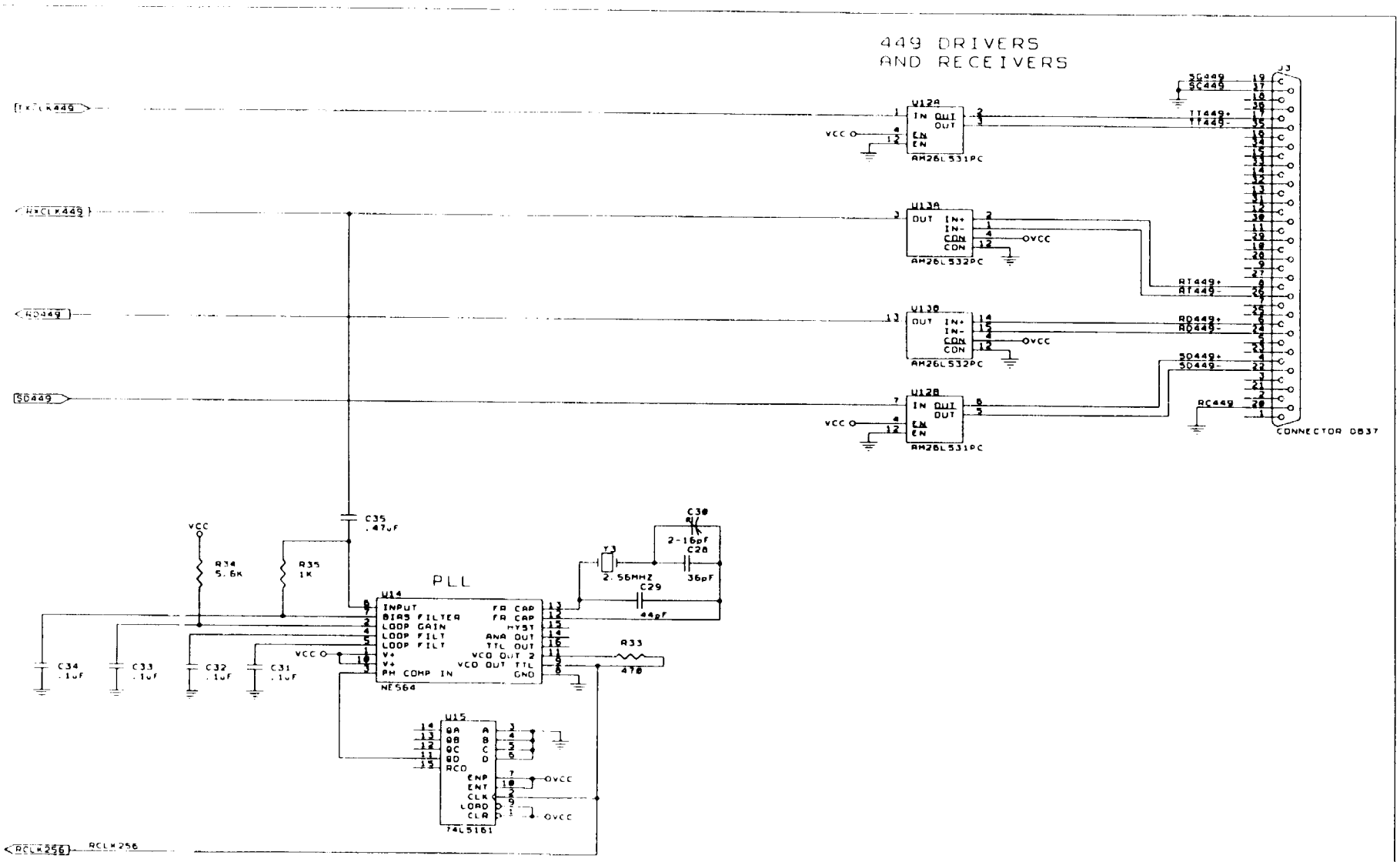
UIFCPU.SCH

	KLT TELECOM INCORPORATED 7230 HIKMAT ROAD MANASSAS, VA 22111		
	U INTERFACE		
	SIZE B	FSCM NO	DWG NO	REV A
March 15, 1992	SCALE		SHEET 2 OF 4



UIFMEM.SCH

	KLT TELECOM INCORPORATED 7238 MIKMAT ROAD MANASSAS, VA 22111		
	U INTERFACE		
	SIZE: B	FSCM NO:	DWG NO: REV: A
March 15, 1992	SCALE:		SHEET 3 OF 4



UIF449.SCH

	XLT TELECOM INCORPORATED 7238 HIKMAT RD. MANASSAS, VA 22111			
	U INTERFACE			
	SIZE: B	FSCM NO	DWG NO	REV: A
March 23, 1992	SCALE		SHEET: 4 OF 4	

Item	Quantity	Reference	Part
1	22	C2,C3,C5,C11,C12,C15,C16, C17,C18,C19,C20,C21,C22, C23,C24,C25,C26,C27,C31, C32,C33,C34	.1uF
2	1	C6	.033uF
3	2	C9,C10	25pF
4	2	C1,C4	.01uF
5	2	C7,C14	100uF
6	1	C8	270pF
7	2	C13,C35	.47uF
8	1	C28	36pF
9	1	C29	44pF
10	1	C30	2-16pF
11	5	D1,D2,D3,D4,D7	1N4001
12	2	D5,D6	VARACTOR
13	4	D8,D9,D10,D11	LED
14	1	J1	RJ-11
15	1	J2	POWER JACK
16	1	J3	HEADER 3
17	1	P1	CONNECTOR DB37
18	5	R1,R6,R7,R12,R35	1K
19	1	R2	10M
20	1	R3	360K
21	2	R4,R5	249K
22	2	R8,R9	14
23	2	R10,R11	1.87K
24	1	R13	3.3K
25	1	R14	700K
26	9	R15,R21,R22,R23,R24,R25, R26,R31,R32	10K
27	2	R16,R17	1M
28	1	R18	1.2K
29	2	R19,R20	4.7K
30	4	R27,R28,R29,R30	180
31	1	R33	470
32	1	R34	5.6K
33	1	T1	U TRANSFORMER
34	1	U1	MC145472
35	1	U2	74LS00
36	1	U3	74LS05
37	1	U4	MC68302
38	1	U5	MC1455
39	2	U6,U8	27C256
40	2	U7,U9	MCM6206
41	1	U10	74LS32
42	2	U11,U13	AM26LS32PC
43	2	U12,U16	AM26LS31PC
44	1	U14	NE564
45	1	U15	74LS161
46	1	Y1	20.48MHZ
47	1	Y2	16.67MHZ
48	1	Y3	2.56MHZ

APPENDIX B

ISTA SOFTWARE SOURCE CODE

This appendix presents an ISTA hardware overview, a review of the design requirements and limitations, list the major software components, and sequences through the software development. A complete listing of the ISTA source code follows a pictorial overview of the "ISTA Software Events and Data Flow Diagram".

CONTENT

ISTA HARDWARE OVERVIEW

- Motorola MC68302 Integrated Multiprotocol Processor (IMP)
- Motorola MC145472 ISDN U-interface Transceiver
- 64K RAM
- 64K EPROM
- LT/NT Selection Switch
- LED Display

DESIGN REQUIREMENTS AND LIMITATIONS

1. U-interface Side

- Operate in either NT or LT mode. Switch selectable.

- Supports only cold start activation mode.

- Only a LT can initiate deactivation process.

- Support Embedded Operation Channel (EOC) processing transparently.

2. Satellite Channel Side

- Supports only HDLC framing in a transparent mode.

- No HDLC protocol processing.

- No retransmission.

- Activation and deactivation conditions conveyed to the remote ISTA.

SOFTWARE COMPONENTS

1. Device Driver

2. U Interface Process (UIPRO)

3. Satellite Channel Process (SATPRO)

4. ISTA Core Process (CORE)

DEVICE DRIVER

1. 68302 Driver

2. U Chip Driver

3. Buffer Manager

4. Interrupt Handler

5. Led Display

DEVICE DRIVER:

68302 Driver

1. 68302 Initialization

- BAR and SCR
- Interrupt Controller and Exception Vector Table.
- Parallel Port A and Port B
    - LED output
    - IDL Clock 2.56 Mhz input
    - SCP enable output
    - U chip reset
    - NT/LT mode input
- SCC1 (for U interface),
    - IDL Non-Multiplexed Serial Interface Mode
    - IDL slave
    - 10-bit mode operation.
    - Total transparent mode (promiscuous mode).
- SCC2 (for satellite interface)
    - HDLC mode operation
- SCP for U-interface control and monitoring

2. Frame Transmission Routines

Frame transmission on SCC and SCP ports

Tx_ui - Send frame out to the U-interface chip

Tx_abort_ui - Abort frame output to U-interface chip

Tx_hdlc - Send data out to the satellite channel

Tx_abort_hdlc - Abort frame output to the satellite channel

Scp_write - write data to SCP port for U-interface chip operation

Scp_read - read data from the U-interface chip control registers.

DEVICE DRIVER:

U CHIP DRIVER (SCP PORT)

1. U Chip Initialization

- LT or NT mode by a hardware switch setting
- 10-bit mode IDL transmission
- IDL master
- All NR3 register interrupts are enabled

2. Channel Activation and Deactivation

U_act_req - for channel activation

U_deact_req - for channel deactivation.

3. Status Monitoring

U_reset - Reset U interface chip

U_eoc_read - Read Embedded Operational Channel (EOC) data

U_m4_read - Read M4 bit information.

DEVICE DRIVER:

BUFFER MANAGER

1. FORMAT

Buffer Header	6
HDLC Address	1
HDLC Control	1
Superframe 2B+D Data	216
M Channel Data	6
HDLC CRC	2
	232

2. SINGLY LINKED LIST

3. OPERATIONAL ROUTINES

Buf_build	-	Initialize a free buffer pool
Buf_get	-	Get free buffer and assign to a SCC receiver
Buf_return	-	Return buffer to free pool

4. SCC Buffer Utilization

- Initially 8 buffers for each SCC's Receive Buffer Descriptor Table.
- Transmit Buffer Descriptor is initially empty.

DEVICE DRIVER:

INTERRUPT HANDLER

1. INTERRUPT SOURCES

- Level 4 Interrupt

  SCC1 and SCC2 Interrupts

- Level 6 Interrupt (EXIRQ)

  U Chip Status Interrupt

2. SCC1 interrupt (U Interface)

- Transparent Mode Events
- Received Buffer
- Receive Busy
- Transmit Buffer
- Transmit Error

3. SCC2 interrupt

- HDLC Mode Events
- Received Frame
- Receive Busy
- Transmit Frame
- Transmit Error

DEVICE DRIVER:

INTERRUPT HANDLER (Continue)

4. U Chip Status Interrupt

   - NR3 register interrupts.
   - link status change (NR1)
   - EOC data update indication
   - end of Superframe indication
   - CRC error indication.

DEVICE DRIVER:

LED Display Routines

Led_on - Turn off a LED.

Led_off - Turn off a LED.

U INTERFACE PROCESS (UIPRO)

FUNCTION:

- U interface activation and deactivation process

- Received U interface Superframe for satellite channel transmission.

- Received satellite link frame for U interface transmission.

- Supplying new data buffer for reception

- Return buffer after transmission for SCC1.

- CRC error handling.

  Notify remote ISTA of CRC error on received Superframe. This frame is transmitted at remote ISTA with forced CRC error at the U interface.

SATELLITE CHANNEL PROCESS (SATPRO)

FUNCTIONS:

- Satellite channel establishment and maintenance.

- 4 HDLC Frame types Process
  - Data Frame
  - Activation Request (AR)
  - Activation Indication (AI)
  - Deactivation Request (DAR).

- Received HDLC frame for U interface transmission.

- Received U interface frame for satellite transmission.

- Supplying new data buffer for reception

- Return buffer after transmission for SCC2.

ISTA CORE PROCESS

FUNCTION:

1. Overall control of the ISTA software execution.

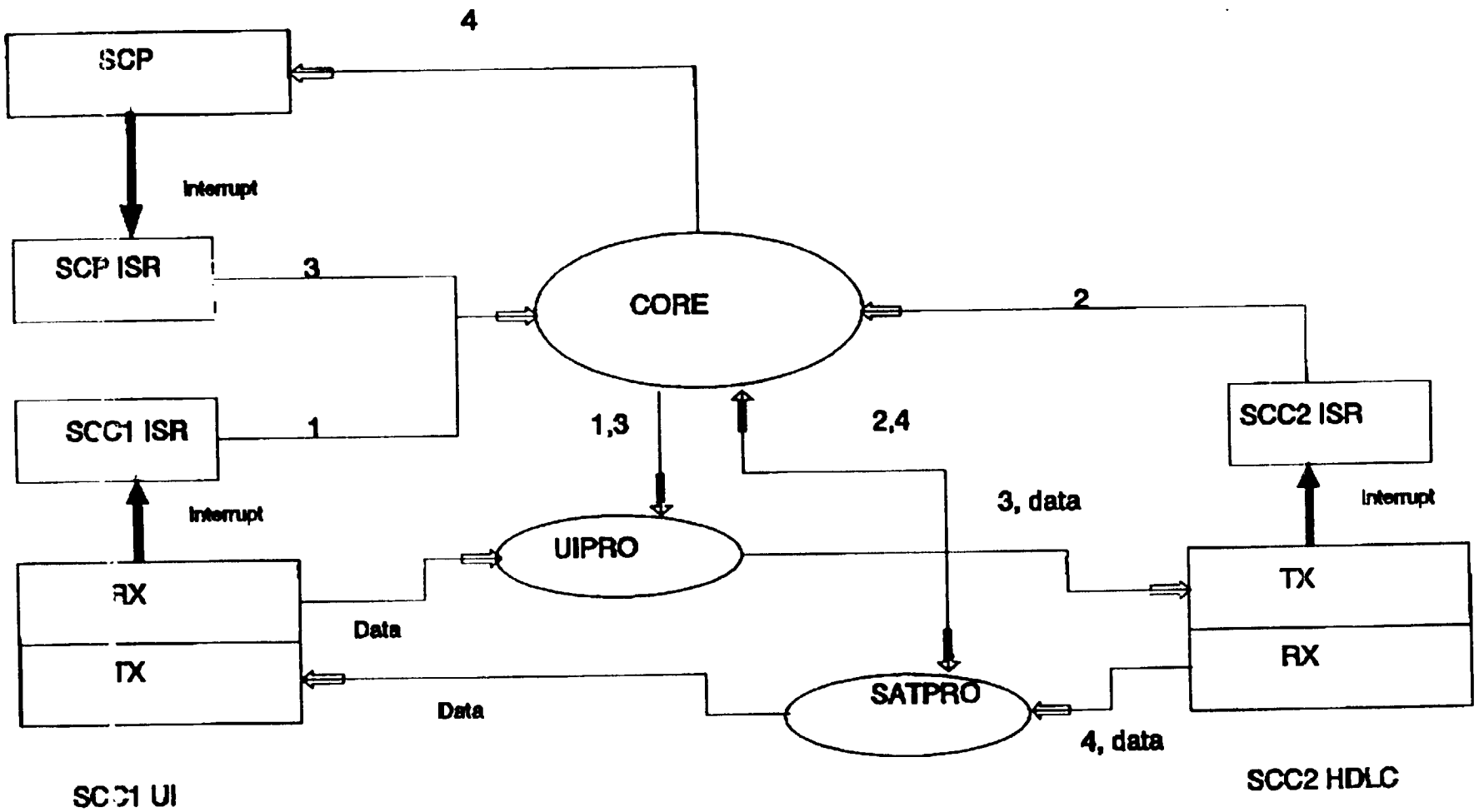
2. Oversees the channel activation, data transmission and deactivation processes at both U interface and satellite interface.

3. Abnormal condition handling.

ISTA STATES

System Reset	-	System power-up or link deactivation.
Ready (idle)	-	Successful system initialization.
Local Activate	-	U-interface Activation at Local ISTA.
Remote Activate	-	Activation request from remote ISTA.
Data Transfer	-	Transparent 2B+D data and EOC channel information transmission
Local Deactivate	-	U interface deactivation at local ISTA.
Remote Deactivate	-	Deactivation Request from remote ISTA
Error State	-	Abnormal condition indicated through LED display.

ISTA Software Events and Data Flow Diagram



1. SCC1 Interrupt Event
   - Rx Buffer
   - Rx Busy
   - Tx Buffer
   - Tx Busy

2. SCC2 Interrupt Event
   - Rx Frame
   - Rx Busy
   - Tx Frame
   - Tx Error

3. SCP Interrupt Event
   - Local AR, DAR
   - EOC Data
   - CRC Error
   - End or SuperFrame

4. Remote ISTA Information
   - Remote AR
   - Remote DAR
   - Remote AI

```
/*
 * bss.h
 * definition of the SCC0 and SCC1 work area
 */

#define CR_WAIT    1000
/*
 * receive error mask bits
 * lower (6) bits are defined as in RX BD status
 */
#define R_BUSY     0x8000

typedef struct work_area {
        unsigned short  rxerr_mask; /* receive error mask */
        BYTE *conf_q;   /* next frame for conf */
        BYTE *tx_q;     /* next frame for tx */
        BYTE *req_q;    /* last requested frame */
        BYTE *rx_q;     /* next frame for rx */
        BYTE *pool_q;   /* first frame in pool */
        BYTE *rtp_q;    /* end of pool (return to pool here) */
        struct scc_pram *pram;     /* port's parameter RAM */
        struct scc_regs *regs;     /* port's SCC registers */
        struct imp_bd   *imp_tbd;  /* next tx BD in table to use */
        struct imp_bd   *imp_cbd;  /* next conf BD in table to use */
        struct imp_bd   *imp_rbd;  /* next BD in table for rx */
        struct imp_bd   *imp_pbd;  /* next BD in table for pool buffer */
        char       tsize;     /* number of Bd's in tx table */
        char       tfree;     /* free BD slots */
        char       rsize;     /* rx table size */
        char       rfree;     /* free BD's in rx table */
        unsigned   tcount;    /* number of frames tx-ed */
        unsigned   rcount;    /* number of frames rx-ed */
        unsigned   terr_cnt;  /* number of transmit errors */
        unsigned   rerr_cnt;  /* number of receive errors */

        unsigned short  fr_lng;    /* max frame length */
        unsigned short  rlen;      /* rx buffer length */

        unsigned short  BDrx;      /* status bits for rx BD's*/

} WORK_AREA;
```

```
/*
 * config.h
 * this file contains definitions
 * for the use of the IMP driver
 * configuration.
 */

/*
 * protocol specific parameter ram
 * initialization.
 */
struct hpram {
        WORD   r_max;           /* maximum frame length */
        WORD   r_mask;          /* address mask */
        WORD   r_addr1;
        WORD   r_addr2;
        WORD   r_addr3;
        WORD   r_addr4;
        WORD   r_cmaskl;        /* CRC constant (low word) */
        WORD   r_cmaskh;        /* CRC constant (high word) */
};

struct driver_config {
        unsigned rom;                   /* base address in ROM */
        struct imp          *internal_ram;      /* address of internal RAM */

        /* configuration control */
        WORD   bar;                     /* base address register */
        LONGWORD       scr;                     /* system control register */

        /* dma */
        WORD   dma_mode;        /* DMA mode register */
        BYTE   dma_fc;          /* DMA function code */

        /* interrupt controller */
        WORD   intr_mode;       /* global intr mode */
        WORD   intr_mask;       /* interrupt mask reg */

        /* parallel port A */
        WORD   pa_control;      /* port A control */
        WORD   pa_direction;    /* port A data direction */
        WORD   pb_control;      /* port B control */
        WORD   pb_direction;    /* port B data direction */

        /* chip selects */
        WORD   cs0_base;        /* base address 0 */
        WORD   cs0_option;      /* option register 0 */
        WORD   cs1_base;
        WORD   cs1_option;
        WORD   cs2_base;
        WORD   cs2_option;
        WORD   cs3_base;
        WORD   cs3_option;

        /* timer */
        WORD   t0_mode;/* timer 1 mode register */
        WORD   t0_ref;          /* timer 1 reference register */
        WORD   t1_mode;/* timer 2 mode register */
```

```
        WORD    t1_ref;             /* timer 2 reference register */
        WORD    wd_ref;             /* watch dog timer reference resist */

        struct scc {
                char    loop_mode;      /* id for loopback (-1 if not) */
                char    upper;
                char    tsize;          /* number of tx buffers */
                char    rsize;          /* number of rx buffers */
                short   minpool; /* min buffers in pool */
                short   psize;          /* # of buffers in pool */
                short   rxerr_mask;     /* receive error mask */
                short   intr_trace;     /* event tracing */

                /* SCC common protocol parameters */
                BYTE    rfc;            /* rx BD FC */
                BYTE    tfc;            /* tx BD FC */
                short           rlen;   /* rx buffer length */

                short           conf;   /* SCC configuration register */
                short           mode;   /* SCC mode register */
                short           sync;   /* SCC sync register */
                short           mask;   /* SCC interrupt mask register */

                /*
                 * protocol specific parameter ram
                 */
                union {
                        struct hpram h;
/* EC debug deleted other mode */
                } pram;

        } scc[3];

        short           scp_smi_mode;   /* SCP/SMI mode register */

        /* physical configuration */
        short           si_mask; /* serial interface mask reg */
        short           si_mode; /* serial interface mode reg */
} DRIVER_CONFIG;
```

```
/*
 * hdlc.h
 * definitions for the HDLC controllers
 */

/*
 * ISTA HDLC frame control filed types
 */
#define  SA_DATA                0x01     /* data frame */
#define SA_AR           0x02    /* Activation Request */
#define SA_AI           0x03    /* Activation Indication */
#define SA_DAR          0x04    /* Deactivation Request */

/*
 * transmit BD's
 */
#define  T_R     0x8000          /* ready bit */
#define  T_X     0x4000          /* extern buffer */
#define  T_W     0x2000          /* wrap bit */
#define  T_I     0x1000          /* interrupt on completion */
#define  T_L     0x0800          /* last in frame */
#define  T_CRC   0x0400          /* transmit CRC (when last) */

#define  T_ERROR         0x000f
#define  T_UN    0x0002          /* error: underrun */
#define  T_CT    0x0001          /* error: CTS lost */

/*
 * receive BD's
 */
#define  R_E     0x8000          /* buffer empty */
#define  R_X     0x4000          /* external buffer */
#define  R_W     0x2000          /* wrap bit */
#define  R_I     0x1000          /* interrupt on reception */
#define  R_L     0x0800          /* last BD in frame */
#define  R_F     0x0400          /* first BD in frame */

#define  R_ERROR         0x00ff
#define  R_LG    0x0020          /* frame too long */
#define  R_NO    0x0010          /* non-octet aligned */
#define  R_AB    0x0008          /* received abort sequence */
#define  R_CR    0x0004          /* receive CRC error */
#define  R_OV    0x0002          /* receive overrun */
#define  R_CD    0x0001          /* carrier detect lost */


/*
 * hdlc interrupts
 */
#define  HDLC_CTS        0x80    /* clear to send changed */
#define  HDLC_CD                 0x40     /* carrier detect changed */
#define  HDLC_IDLE       0x20    /* idle sequence status changed */
#define  HDLC_TXE        0x10    /* transmit error */
#define  HDLC_RXFR       0x08    /* receive frame */
#define  HDLC_BUSY       0x04    /* busy */
#define  HDLC_TXBD       0x02    /* transmit buffer */
#define  HDLC_RXBD       0x01    /* receive buffer */
```

```
/*
 * TRANSPARENT MODE INTERRUPT
 */
#define  TP_TXE 0x10     /* transmit error */
#define  TP_RXCH         0x08     /* receive character, a word */
#define  TP_BUSY         0x04     /* busy */
#define  TP_TXBD         0x02     /* transmit buffer */
#define  TP_RXBD         0x01     /* receive buffer */
```

```
/*********************************************************
 * File:   hdlc_r.c
 *
 * Description:
 *         The driver routines for handling receive
 *         on an HDLC port.
 *
 * Routines:
 *         hdlc_rxfr
 *         hdlc_busy
 *
 * Author:
 *         Enghe Chimood
 *********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "hdlc.h"
#include "imp.h"

/*********************************************************
 * routine:          hdlc_rxfr
 *
 * description:
 *         This is reached after being sent a RX_IND mesage
 *         by the interrupt routine (interrupt on IMP
 *         receiving an HDLC frame).
 *         The receive buffer table is flushed of all HDLC
 *         frames received: these frames are sent to the
 *         port's upper layer.
 *         The message sent is only a 32-bit code (not an
 *         allocated MSG structure) and therefore must
 *         not be relm-ed. Its structure corresponds to
 *         the first 32-bits of a normal HDR structure.
 *
 * arguments:
 *         h                    the event message received
 *
 * return code:
 *
 * side effects:
 *         Advance imp_rbd each time a
 *         received buffer is collected.
 *         The rxfr interrupt must be re-enabled after
 *         all received frames have been collected.
 *
 *********************************************************/
hdlc_rxfr(h, child)
HDR *h;
CHILD *child;
{
        CCT *cct;
        register WORK_AREA *wa;
        register unsigned short status;                 /* status of BD */
        register R_FRAME *r1, *r2;
                        /* r1 = first r_frame in frame */
```

```
                        /* r2 = current r_frame */
register R_FRAME *r3;
                        /* r3 = previous r_frame */
register IMP_BD *nextBD;  /* next BD in frame */
int flen;  /* accumulative frame length */
int plen;  /* previous frame length (till last buffer) */
char eofr;              /* boolean: end of frame recognized */
char bcount;            /* number of BDs in this frame */
int crc;   /* number of bytes in crc */
struct hdlc_mode_reg *mode;

GETGCT(gct);
wa = (WORK_AREA *)child->work_area;

/*
 * clear event bit
 * this clears indications of old (masked) events
 */
wa->regs->event = (HDLC_RXFR | HDLC_BUSY);

/*
 * if receive is disabled,
 * return immediately
 */
if( wa->flags & RX_DISABLE )
        if( wa->flags & RX_ACTIVATE )
                wa->flags &= ~RX_ACTIVATE;
        else
                return;

if( !wa->rx_q->nextb ){
        wa->regs->mask |= HDLC_RXFR;
        return;
}

/*
 * rbd address next BD for receive
 */
nextBD = wa->imp_rbd;
status = nextBD->status;
r1 = r2 = r3 = wa->rx_q;
flen = 0;
eofr = bcount = 0;
while( !(status & R_E) ){
        /*
         * first check that r2 has been
         * linked to the rx BD table
         */
        if( !(r2->flags & LINKED_TO_TABLE) )
                break;

        bcount++;

        /*
         * if first is set on the not first
         * frame, the last buffer(s) where lost (busy?)
         * and r2 is now start of new frame
         */
```

```
if( (r1 != r2) && (status & R_F) ){
        r1->flen = plen;
        r1->hdr.status = RECEIVE_ERROR;
        r1->hdr.err_info = R_BUSY;
        r3->nextb = 0;
        wa->rerr_cnt++;
        driver_trace(RX_IND, h->id, r1);
        /* send frame to upper layer */
        if( R_BUSY & ~wa->rxerr_mask )
                return_to_pool(r1, child);
        else
                if( (*gct->send)(child->upper, r1) )
                        relm(r1);

        /*
         * adjust accumulative
         * frame lengths
         */
        flen = nextBD->length;
        plen = 0;

        /* advance frame pointer */
        r1 = r2;
        eofr = 1;
}


/*
 * update accumulative frame length
 * on the last BD in a frame, the length
 * is the length of the entire HDLC frame.
 */
plen = flen;
if( status & R_L ){
        flen = nextBD->length;
        /*
         * remove the CRC from valid frames
         * (note that the CRC may be split between
         * the last two buffers)
         */
        if( !(status & R_ERROR) ){
                mode = (struct hdlc_mode_reg *)
                        &wa->regs->mode;
                crc = (mode->crc_32? 4: 2);
                flen -= crc;
        }
        if( status & R_LG )
                r2->blen = wa->fr_lng - plen;
        else if( flen >= plen )
                /* CRC in last buffer */
                r2->blen = flen - plen;
        else {
                /* some of CRC in second last buffer */
                r2->blen = 0;
                r3->blen -= (plen - flen);
                /* release emptied buffer */
                r3->nextb = r2->nextb;
                r2->nextb = 0;
```

```
                relm(r2);
                r2 = r3;
        }
} else {
        flen += nextBD->length;
        r2->blen = nextBD->length;
}


/*
 * if end of frame ...
 */
if( status & R_L ){
        /*
         * update status and length
         * in new received frame,
         * zero accumulative length
         * for next frame
         */
        r1->flen = flen;    /* frame total length */
        r3 = r2;
        r2 = r2->nextb;
        r3->nextb = 0;      /* no more buffs in this frame */

        /*
         * prepare for next R_FRAME
         */
        flen = 0;

        driver_trace(RX_IND, h->id, r1);
        /*
         * pass r_frame to upper layer
         * (subject to error mask).
         * buffers are linked via nextb
         */
        if( status & R_ERROR ){
                driver_trace(RX_ERROR,
                        r1->hdr.id, nextBD);
                r1->hdr.status = RECEIVE_ERROR;
                r1->hdr.err_info = status & R_ERROR;
                wa->rerr_cnt++;
                if( (status&R_ERROR) & ~wa->rxerr_mask )
                        return_to_pool(r1, child);
                else
                        if( (*gct->send)(child->upper, r1) )
                                relm(r1);
        } else
                if( (*gct->send)(child->upper, r1) )
                        relm(r1);
        r1 = r3 = r2;
        wa->rcount++;
        eofr = 1;
} else {
        r3 = r2;
        r2 = r2->nextb;     /* next R_FRAME for rx */
}

/*
```

```
                * advance rbd (to next Rx BD in table)
                */
                if( status & R_W )
                        nextBD = wa->pram->rbd;
                else
                        nextBD++;

                if( eofr ){
                        wa->rfree += bcount;
                        eofr = bcount = 0;
                        wa->imp_rbd = nextBD;
                }

                status = nextBD->status;        /* update status */
        }

        wa->rx_q = rl;

        /*
         * replenish the rx BD table
         */
        status = replenish_rxbd(wa);
        if( status ){
                MSG *m;
                extern MSG *getm();

                driver_trace(BUSY_IND, h->id, status);
                m = getm(BUSY_IND, h->id, 0, 0);
                if( m ){
                        m->hdr.status = status;
                        if( (*gct->send)( child->upper, m) )
                                relm(m);
                }
                /*
                 * enable BUSY interrupts if
                 * local busy was just cleared
                 */
                if( status == LOCAL_BUSY_CLEARED )
                        wa->regs->mask |= HDLC_BUSY;
        }

        /*
         * re-enable interrupts
         * setting the mask will result in an interrupt
         * if any frames were received during the time
         * the mask was zero. therefore, frames will not
         * go unnoticed if received after exiting the while
         * loop above and before the mask bit is set.
         */
        wa->regs->mask |= HDLC_RXFR;
}


/*****************************************************
 * routine:         hdlc_busy
 *
 * description:
 *      Notify the upper layer that the
```

```
 *          busy condition has been encountered.
 *
 * arguments:
 *          h                   points to dummy (32 bit) header
 *          child               child structure
 *
 * return code:
 *
 * side effects:
 *
 ************************************************************/
hdlc_busy(h, child)
HDR *h;
CHILD *child;
{
        GCT *gct;
        MSG *m;
        WORK_AREA *wa;
        extern MSG *getm();

        /*
         * mark local busy
         * (checked when returning to pool)
         */
        wa = (WORK_AREA *)child->work_area;
        wa->flags |= LOCAL_BUSY;

        /*
         * NOTE: this should be redundant
         */
        hdlc_rxfr(h, child);

        /*
         * if still busy, notify upper layer
         */
        if( wa->flags & (LOCAL_BUSY | POOL_EMPTY) ){
                GETGCT(gct);
                driver_trace(BUSY_IND, h->id, LOCAL_BUSY_DETECTED);
                m = getm(BUSY_IND, h->id, 0, 0);
                if( m ){
                        m->hdr.status = LOCAL_BUSY_DETECTED;
                        if( (*gct->send)(child->upper, m) )
                                relm(m);
                }
        }
}
```

```
/*******************************************************
 * File:    hdlc_t.c
 *
 * Description:
 *  The driver routines for handling
 *  transmission on an HDLC port.
 *
 * Routines:
 *  hdlc_txreq
 *  hdlc_conf
 *  hdlc_txerr
 *  hdlc_stoptx
 *
 * Author:
 *  Enghe Chimood
 *******************************************************/

#include "system.h"
#include "config.h"
#include "hdlc.h"
#include "register.h"
#include "imp.h"

/*******************************************************
 * routine: hdlc_txreq
 *
 * description:
 *  Request for transmission of an HDLC channel.
 *  T_FRAMEs are already linked via the driver's
 *  nextf pointer. It holds them in a linked list,
 *  linking them to the IMP's BD table as slots
 *  become available.
 *
 * arguments:
 *  t       points to first T_FRAME for tx
 *
 * return code:
 *
 * side effects:
 *******************************************************/

hdlc_txreq(t, child)
register T_FRAME *t;
CHILD *child;
{
        register WORK_AREA *wa;

        wa = (WORK_AREA *)child->work_area;
        if( !wa->req_q )
                wa->tx_q = wa->conf_q = t;
        else {
                wa->req_q->layer[0].nextf = t;
                if( !wa->tx_q )
                        wa->tx_q = t;
        }

        /*
```

```
        * find new end of queue
        */
        while( t->layer[0].nextf )
                t = t->layer[0].nextf;
        wa->req_q = t;

        transmit_frames(wa);
}


/*********************************************************
 * routine: hdlc_conf
 *
 * description:
 *  Confirm transmitted frames. This routine
 *  is reached as result of a TX_CONF (primitive)
 *  message sent to the driver module by the
 *  SCC interrupt routine handler.
 *  It implies that transmission of an HDLC
 *  frame has been completed.
 *  Note that transmit_frames is called to possibly link
 *  new frames to the BD entries that have been
 *  released.
 *  The message sent is only a 32-bit code (not an
 *  allocated MSG structure) and therefore must
 *  not be relm-ed. Its structure corresponds to
 *  the first 32-bits of a normal HDR structure.
 *
 * arguments:
 *  h       the event message received
 *  child       the child structure involved
 *
 * return code:
 *
 * side effects:
 *  Interrupts for TXBD should be re-enabled (they
 *  were disabled in the interrupt routine).
 *
 *********************************************************/
hdlc_conf(h, child)
HDR *h;
CHILD *child;
{
        GCT *gct;
        register WORK_AREA *wa;
        register int i = 0;
        T_FRAME *t;
        extern T_FRAME *confirm_frame();

        GETGCT(gct);
        wa = (WORK_AREA *)child->work_area;

        /*
         * clear event bit
         * this clears indications of old (masked) events
         */
        wa->regs->event = HDLC_TXBD;
```

```
        /*
         * if confirm is disabled,
         * return immediately
         */
        if( wa->flags & CONF_DISABLE )
                if( wa->flags & CONF_ACTIVATE )
                        wa->flags &= ~CONF_ACTIVATE;
                else
                        return;

        while( t = confirm_frame(wa) ){
                i++;
                driver_trace(TX_CONF, h->id, t);
                conf_frm(t, child->upper);
        }

        /*
         * re-enable interrupts before calling transmit_frames!!
         */
        wa->regs->mask |= HDLC_TXBD;

        if( i )
                transmit_frames(wa);
}


/*********************************************************
 * routine: hdlc_txerr
 *
 * description:
 *  Called after receiving an HDLC_TXE interrupt.
 *  This can result from either CTS lost or
 *  transmit underrun.
 *  The message sent is only a 32-bit code (not an
 *  allocated MSG structure) and therefore must
 *  not be relm-ed. Its structure corresponds to
 *  the first 32-bits of a normal HDR structure.
 *
 * arguments:
 *  h       the event message received
 *  child       the child structure concerned
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
hdlc_txerr(h, child)
HDR *h;
CHILD *child;
{
        WORK_AREA *wa;

        wa = (WORK_AREA *)child->work_area;

        /*
         * clean up after error
```

```
        */
        transmit_error(wa);

        /*
         * confirm all transmitted frames
         */
        hdlc_conf(h, child);
}

/*********************************************************
 * routine: hdlc_abort
 *
 * description:
 *  Sent by an upper layer to immediately
 *  stop transmitting on this channel
 *  (if a buffer is currently being tx-ed,
 *  an abort frame will be generated).
 *  The STOP_TX command is issued.
 *
 * arguments:
 *  h       the event message received
 *  child       the child structure concerned
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
hdlc_abort(h, child )
HDR *h;
CHILD *child;
{
        WORK_AREA *wa;

        wa =(WORK_AREA *)child->work_area;
        issue_cmd(STOP_TX + (h->id << 1));
        wa->flags |= TX_STOPPED;
        wa->restart = RESTART_TX + (h->id << 1);
        relm(h);
}
```

```
/*
 * imp.h
 * definitions of IMP memory structures
 */

/* Buffer Descriptors */
typedef struct imp_bd {
        WORD   status;
        WORD   length;
        BYTE   *buf;
} IMP_BD;

/*
 * transmit BD's status (common to all modes)
 */
#define T_R 0x8000     /* ready bit */
#define T_X 0x4000     /* extern buffer */
#define T_W 0x2000     /* wrap bit */
#define T_I 0x1000     /* interrupt on completion */

/*
 * receive BD's status (common to all modes)
 */
#define R_E 0x8000     /* buffer empty */
#define R_X 0x4000     /* external buffer */
#define R_W 0x2000     /* wrap bit */
#define R_I 0x1000     /* interrupt on reception */


/*********************************************************************
        HDLC parameter RAM
*********************************************************************/

struct hdlc_pram {
        BYTE   rfc;      /* Rx FC register */
        BYTE   tfc;      /* Tx FC register */
        WORD   rblen;    /* Rx buffer length */
        WORD   rstate;   /* Rx internal state */
        WORD   rbuf;     /* Rx buffer number */
        LONGWORD   rptr;     /* Rx internal data pointer */
        WORD   rcount;   /* Rx byte count */
        WORD   rtemp;    /* Rx temp */

        WORD   tstate;   /* Tx internal state */
        WORD   tbuf;     /* Tx buffer number */
        LONGWORD   tptr;     /* Tx internal data pointer */
        WORD   tcount;   /* Tx byte count */
        WORD   ttemp;    /* Tx temp */

        WORD   rcrc1;    /* CRC low word (Rx) */
        WORD   rcrc2;    /* CRC high word (Rx) */
        WORD   r_cmaskl;  /* constant */
        WORD   r_cmaskh;  /* constant */
        WORD   tcrc_l;    /* CRC low word (Tx) */
        WORD   tcrc_h;    /* CRC high word (Tx) */

        WORD   discard_cnt;   /* discarded frames counter */
        WORD   crc_cnt;   /* CRC error counter */
```

```
        WORD    abort_cnt;  /* rx-ed abort counter */
        WORD    nmf_cnt;    /* not my frame counter */
        WORD    retx_cnt;   /* retransmission counter */

        WORD    r_max;      /* maximum frame length */
        WORD    r_maxcnt;   /* maximum frame counter */
        WORD    r_mask;     /* address mask */
        WORD    r_addr1;    /* address 1 */
        WORD    r_addr2;    /* address 2 */
        WORD    r_addr3;    /* address 3 */
        WORD    r_addr4;    /* address 4 */
} HDLC_PRAM;


/*****************************************************************
        Transparent Mode parameter RAM
*****************************************************************/

struct txparent_pram {
        BYTE    rfc;        /* Rx FC register */
        BYTE    tfc;        /* Tx FC register */
        WORD    rblen;      /* Rx buffer length */
        WORD    rstate;     /* Rx internal state */
        WORD    rbuf;       /* Rx buffer number */
        LONGWORD    rptr;       /* Rx internal data pointer */
        WORD    rcount;     /* Rx byte count */
        WORD    rtemp;      /* Rx temp */

        WORD    tstate;     /* Tx internal state */
        WORD    tbuf;       /* Tx buffer number */
        LONGWORD    tptr;       /* Tx internal data pointer */
        WORD    tcount;     /* Tx byte count */
        WORD    ttemp;      /* Tx temp */

        WORD    reserve[0x12];
} TXPARENT_PRAM;

/*****************************************************************
        SCP/SMI parameter RAM
        (overlaid on tx bd[6,7] of SCC channel[2])
*****************************************************************/

#define SCP_R   0x8000      /* Ready bit in BD */

struct scp_smi_pram {
        WORD    res[3];     /* reserved */
        WORD    smi0_r;     /* smi 0 Rx BD */
        WORD    smi0_t;     /* smi 0 Tx BD */
        WORD    smi1_r;     /* smi 1 Rx BD */
        WORD    smi1_t;     /* smi 1 Tx BD */
        WORD    internal[6];    /* for internal use only */
        WORD    scp_bd;     /* Tx/Rx BD */
        WORD    berr;       /* bus error channel number */
        WORD    rev_number; /* chip revision number */
} SCP_SMI_PRAM, *P_SCP_SMI_PRAM;

/*
```

```
 * internal SYSTEM ram
 */
typedef struct imp {
/* BASE + 0x000: user data memory */
        BYTE   udata[0x240];  /* 0x240 bytes user data */
        BYTE   uempty[0x1c0]; /* empty till 0x400 */

/* BASE + 0x400: PARAMETER RAM */
        union {
                struct scc_pram {
                        IMP_BD  rbd[8];
                        IMP_BD  tbd[8];
                        union {
                                HDLC_PRAM   h;
                                TXPARENT_PRAM   t;
                                BYTE        sc[0x40];
                        } pscc;    /* scc parameter area (protocol dependent) */
                } scc;
                BYTE   pr[0x100];
        } pram[3];
/* reserved */
        BYTE   resrvd1[0x100];

/* BASE + 0x800: INTERNAL REGISTERS */
/* DMA */
        WORD    dma_res1;  /* reserved */
        WORD    dma_mode;  /* dma mode reg */
        LONGWORD    dma_s;     /* dma source */
        LONGWORD    dma_d;     /* dma destination */
        WORD    dma_cnt;   /* dma byte count */
        BYTE    dma_status; /* dma status */
        BYTE    dma_res2;  /* reserved */
        BYTE    dma_fc;    /* dma function code */
        BYTE    dma_res3;  /* reserved */
/* Interrupt Controller */
        WORD    intr_mode; /* interrupt mode register */
        WORD    intr_pending;  /* interrupt pending register */
        WORD    intr_mask; /* interrupt mask register */
        WORD    intr_inservice; /* interrupt in service register */
        LONGWORD    intr_res;  /* reserved */
/* Parallel I/O */
        WORD    pa_control; /* port A control */
        WORD    pa_direction;  /* port A data direction */
        WORD    pa_data;   /* port A data register */
#define   SCP_EN_MSK  0x080
#define   U_RESET_MSK 0x100
#define   NT_MODE_MSK 0x200

        WORD    pb_control; /* port B control */
        WORD    pb_direction;  /* port B data direction */
        WORD    pb_data;   /* port B data */
#define USYNC_LED_MSK  0x01
#define SATSYNC_LED_MSK 0x02
#define OTHER_LED_MSK  0x04

        WORD    p_res;     /* reserved */
/* Chip Select */
        LONGWORD    cs_res;
```

```
        WORD   cs0_base;  /* base address 0 */
        WORD   cs0_option; /* option register 0 */
        WORD   cs1_base;
        WORD   cs1_option;
        WORD   cs2_base;
        WORD   cs2_option;
        WORD   cs3_base;
        WORD   cs3_option;
/* Timer */
        WORD   t0_mode;   /* timer 0 mode register */
        WORD   t0_ref;    /* timer 0 reference register */
        WORD   t0_cap;    /* timer 0 capture register */
        WORD   t0_counter; /* timer 0 counter */
        BYTE   tim_res0;  /* reserved */
        BYTE   t0_status;  /* timer 0 status */

        WORD   wd_ref;    /* watch dog timer reference */
        WORD   wd_counter; /* watch dog counter */
        WORD   tim_res2;  /* reserved */

        WORD   t1_mode;   /* timer 1 mode register */
        WORD   t1_ref;    /* timer 1 reference register */
        WORD   t1_cap;    /* timer 1 capture register */
        WORD   t1_counter; /* timer 1 counter */
        BYTE   tim_res3;  /* reserved */
        BYTE   t1_status;  /* timer 1 status */
        WORD   tim_res4[3];   /* reserved */
/* command register */
        BYTE   cr;    /* command register */
        BYTE   cr_res;    /* reserved */
/* reserved */
        BYTE   resrvd2[0x1e];
/* SCC registers */
        struct scc_regs {
                WORD   resvd;  /* reserved */
                WORD   conf;   /* SCC configuration register */
                WORD   mode;   /* SCC mode register */
                WORD   sync;   /* SCC sync register */
                BYTE   event;  /* SCC event register */
                BYTE   res1;   /* reserved */
                BYTE   mask;   /* SCC mask register */
                BYTE   res2;   /* reserved */
                BYTE   status; /* SCC status register */
                BYTE   res3;   /* reserved */
                WORD   res4;   /* reserved */
        } scc_regs[3];
/* SP (SCP + SMI) */
        WORD   sp_scm;    /* scp, smi mode + clock control */
/* Serial Interface */
        WORD   si_mask;   /* mask register */
        WORD   si_mode;   /* mode register */

/* reserved */
        BYTE   resrvd3[0x7a],
} IMP;
```

```
/*********************************************************
 * File:    intr.c
 *
 * Description:
 *  IMP interrupt routines.
 *  These should all be called from user code
 *  via the GCT.
 *  All routines expect to be passed a pointer to
 *  internal RAM as a parameter.
 *
 * Routines:
 *  u_intr
 *  scc0_intr
 *  scc1_intr
 *  scc2_intr
 *  scp_intr
 *  error_intr
 *  scc_intr
 *
 * Author:
 *  Enghe Chimood
 *********************************************************/
#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"

#define CLEAR_BIT(x, bit)   x = bit

static intr_event (LONGWORD event);

u_intr()   /* IRQ6 interrupt */
{
BYTE intr;
BYTE nr1_data, br1_data, br3_data;
WORD nr6_data;
INTR_EVENT tmp;

        /* read interrupt status register */
        scp_nb_read (NR3, &intr);

        if (intr & U_IRQ3)
        {
                scp_nb_read (NR1, &nr1_data);
                tmp.u.ui.u_reg = (WORD) nr1_data;
                tmp.u.ui.evt = U_IRQ3;
                intr_event (tmp.u.lw);
        }

        if (intr & U_IRQ2)
        {
                scp_nr6_read (NR6, &nr6_data);
                tmp.u.ui.u_reg = nr6_data;
                tmp.u.ui.evt = U_IRQ2;
                intr_event (tmp.u.lw);
        }
        if (intr & U_IRQ1)
        {
```

```
                scp_br_read (BR1, &br1_data);
                tmp.u.ui.u_reg = (WORD) br1_data;
                tmp.u.ui.evt = U_IRQ1;
                intr_event (tmp.u.lw);
        }

        if (intr & U_IRQ0)
        {
                scp_br_read (BR3, &br3_data);
                tmp.u.ui.u_reg = (WORD) br3_data;
                tmp.u.ui.evt = U_IRQ0;
                intr_event (tmp.u.lw);
        }

        CLEAR_IPA;
}

scc0_intr()
{
        scc_intr(0);
        CLEAR_BIT(Imp->intr_inservice, INTR_SCC0);
        CLEAR_IPA;
}

scc1_intr()
{
        scc_intr(1);
        CLEAR_BIT(Imp->intr_inservice, INTR_SCC1);
        CLEAR_IPA;
}

scc2_intr()
{
        CLEAR_BIT(Imp->intr_inservice, INTR_SCC2);
        scc_intr(2);
        CLEAR_IPA;
}

scp_intr()
{
        CLEAR_BIT(Imp->intr_inservice, INTR_SCP);
        (*gct->send)(DRIVER, &wa->rxind_event);
        CLEAR_IPA;
}

error_intr()
{
        CLEAR_BIT(Imp->intr_inservice, INTR_ERR);
        CLEAR_IPA;
}

/*******************************************************
 * routine: scc_intr
 *
 * description:
 *  Called by the sccN_intr routine.
 *  Sends the appropriate message to the ISTA Core process.
 *  A message from an interrupt routine consists
```

```
*  of a 32-bit number (NOT a pointer to a MSG)
*  defined as follows:
*     16 MSBs = interrupt_tye
*     16 LSBs = channel number

*  a proper MSG structure is sent only to the SYS_MNG module
*  for non-primitive event notification.
*  Note: Two variables are used to store the event register.
*  One (event) is used for event notification: all masked
*  event bits are cleared here before examining its contents.
*
* arguments:
*  port       the channel number (0 - 2)
*
* return code:
*
* side effects:
*
*******************************************************/

scc_intr(port)
BYTE port;
{
        register unsigned char event;
        register struct scc_regs *regs;
        int j, cmd;
        register IMP_BD *bd1, *bd2;

        regs = &Imp->scc_regs[port];
        event = regs->event;
        /* only consider unmasked interrupts */
        event &= regs->mask;

        if (port == 0)     /* U interface port */
        {
        /*
         * send event notifications.
         * only issue the protocol interrupts once:
         * the module will re-enable them
         */
                if( event & TP_TXBD ){
                        regs->mask &= ~TP_TXBD;
                        intr_event (TP_TXBD << 16 | port);
                }

                if( event & TP_RXBD ){

                        regs->mask &= ~TP_RXBD;
                        intr_event (TP_RXBD << 16 | port);
                }

                if( event & TP_BUSY ){
                        regs->mask &= ~TP_BUSY;
                        intr_event (TP_BUSY << 16 | port);
                }

                if( event & TP_TXE )
                        intr_event (TP_TXE << 16 | port);
```

```
        }

        if (port == 1)  /* HDLC port */
        {
                if( event & HDLC_TXBD ){
                        regs->mask &= ~HDLC_TXBD;
                        intr_event (HDLC_TXBD << 16 | port);
                }

                if( event & HDLC_RXFR ){

                        regs->mask &= ~HDLC_RXBD;
                        intr_event (HDLC_RXBD << 16 | port);
                }

                if( event & HDLC_BUSY ){
                        regs->mask &= ~HDLC_BUSY;
                        intr_event (HDLC_BUSY << 16 | port);
                }

                if( event & HDLC_TXE )
                        intr_event (HDLC_TXE << 16 | port);

        }
        /*
         * cleared handled event bits
         */
        CLEAR_BIT(regs->event, event);
}

static intr_event (event)
LONGWORD event;
{
        Intr_evt[Head_idx] = (INTR_EVENT) event;
        Head_idx = (++head_idx) % MAX_EVT_ENTRY;
@
```

```
/************
** MODULE
**   ISTA Buffer Management
** ----------
** FILE NAME
**   ISTABUF.C
** ----------
** DESCRIPTION
**
**  This file contains buffer management routines for the ISTA software
**
** ----------
** REVISION HISTORY
**
**   1. 03/02/92   Enghe Chimood
**        Created original
**
** ----------
** NOTES
**
************/

#include "system.h"
#include "istabuf.h"          /* buffer header definitions */

/* GLOBAL VARIABLES */

extern QHEAD    Free_q;
extern BYTE Buffer_area[FRAME_SIZE * BUF_COUNT];

/*****************************************************************************
**                BUFFER MANAGEMENT CONTROL BLOCK
*****************************************************************************/

void Ista_init_bufmnt (void)

{
        /*
        ** Build Freeq Q
        */

        Ista_define_q (&Free_q);

        Ista_build_q
                        (&Free_q,
                        FRAME_SIZE - BUF_HEADER_SIZE,
                        (LONGWORD) Buffer_area,
                        FRAME_SIZE * BUF_COUNT);
}

/************
** FUNCTION
**   Build a queue of free buffers
** ---------
*/
          void  Ista_build_q
/*
** DESCRIPTION
```

```
**
**   Ista_build_q routine is used to build a queue of free
**   buffers according to the specified buffer size from the
**   specified memory area.
**
** INPUT PARAMETERS
*/
        (
                                /* the address of the queue header */
        p_QHEAD   p_queue,

                                /* data size of the buffer to be built. */
        WORD      data_size,

                                /* memory starting address from where buffers
                                 * are to be built. */
        LONGWORD  mem_addr,

                                /* size of the memory */
        WORD      mem_size
        )
/*
** OUTPUT PARAMETERS
**
**   None
**
** RETURN VALUES
**
**   None
**
** ----------
** NOTES
**
**    1. The p_queue queue header needs to be "defined" previously.
**
************/
{
        p_BUF_HEADER p_buf;
        WORD       total_buf_size;
        LONGWORD   end_mem_addr;

        /* Compute the total number of bytes in the buffer */
        total_buf_size = BUF_HEADER_SIZE + data_size;

        end_mem_addr = mem_addr + mem_size;

        /* Keep going as long as there is enough memory */
        while (( mem_addr < end_mem_addr ) &&
                  ( mem_size >= total_buf_size ))
        {
                p_buf = (p_BUF_HEADER) mem_addr;

                /* initialize the buffer header and put it on the queue */
                p_buf->p_next = NULL;
                p_buf->data_count = 0;

                Nq (p_queue, &p_buf);
```

```
                /* move to the next buffer address */
                mem_size -= total_buf_size;
                mem_addr += total_buf_size;
        }
}

/************
** FUNCTION
**   Define a queue header
** ----------
*/
        void  Ista_define_q
/*
** DESCRIPTION
**
**   This function is used to define queue specified.  It initialize
**   the buffer pointers and buffer count.
**
** INPUT PARAMETERS
*/
          (
                                        /* the address of the queue header */
          p_QHEAD  p_queue,

/*
** OUTPUT PARAMETERS
*/
          )
/*
** RETURN VALUES
**
**   None
**
** ----------
** NOTES
**
************/
{
        p_queue->p_next = NULL;
        p_queue->p_last = NULL;

        /* There are no buffers on the queue */
        p_queue->queue_depth = 0;
}

/************
** FUNCTION
**   Return a buffer to free queue
** ----------
*/
  STATUS_CODE  Ista_return_buf
/*
** DESCRIPTION
**
**   Return_buf is used to return a buffer to its original
**   free queue.  If the buffer count is incremented to or beyond
**   the high mark, the caller is informed through the return
**   value.
```

```
**
**   The pointer passed MUST NOT be positioned at the start of the
**   BUF_HEADER.
**
** INPUT PARAMETERS
*/
        (
                        /* address of the "buffer pointer" to be returned */
          p_BYTE  *p_buffer

/*
** OUTPUT PARAMETERS
**
**   pointer to buffer is NULL.
*/

        )
/*
** RETURN VALUES
**
**   S_OK    - the buffer is returned to its original queue
**
** ----------
** NOTES
**
**
************/
{
        p_BUF_HEADER p_buf_hdr, p_temp_buf;

        /* type casting, but not ready to be used yet */
        p_buf_hdr = (p_BUF_HEADER) (*p_buffer);

        /* move back to the start of BUF_HEADER */
        p_buf_hdr--;

        /* this is the original caller's variable */
        *p_buffer = NULL;

        /* zero out the actual data byte count of the buffer */
        p_buf_hdr->data_count = 0;

        /* Since this is the last buffer on the queue, it points to no one */
        p_buf_hdr->p_next = NULL;

        /* if the queue is empty, point the head and tail to the new buffer */
        if ( Free_q.p_next == NULL )
        {
                Free_q.p_next = p_buf_hdr;
                Free_q.p_last = p_buf_hdr;
        }
        else   /* attach the buffer to the end */
        {
                p_temp_buf = Free_q.p_last;
                p_temp_buf->p_next = p_buf_hdr;
                Free_q.p_last = p_buf_hdr;
        }
```

```
        /* Increment the number of entries in the queue */
        Free_q.queue_depth++;

        return (S_OK);
}

/************
** FUNCTION
**   Get a buffer from Free queue
** ----------
*/
        STATUS_CODE  Ista_get_buf
/*
** DESCRIPTION
**
**   This function is used to get a buffer from the head of the
**   specified queue.
**
** INPUT PARAMETERS
*/
        (
/*
** OUTPUT PARAMETERS
*/
                                                /* address of the "buffer pointer" if one has been
                                                 * dequeued
                                                 */
         p_BYTE      *p_buffer

        )
/*
** RETURN VALUES
**
**   S_OK    - buffer is dequeued
**   S_Q_EMPTY      - queue is empty and NO buffer is dequeued
**
** ----------
** NOTES
**
**
************/
{
p_BUF_HEADER p_buf_hdr;

*p_buffer = NULL;

        /* If the queue is empty, let's not prolong the agony... */
        if (Free_q.queue_depth == 0)
                return (S_Q_EMPTY);

        /* now, dequeue a buffer */
        p_buf_hdr = Free_q.p_next;
        *p_buffer = (p_BYTE) p_buf_hdr;
        Free_q.p_next = p_buf_hdr->p_next;

        /* Is this the only buffer on the queue? */
        if (Free_q.p_next == NULL)
```

```
            Free_q.p_last = NULL;

        /* NILL the next buffer pointer of the dequeued buffer */
        p_buf_hdr->p_next = NULL;

        /* Decrement the number of entries in the queue */
        Free_q.queue_depth --;

        /* skip over the BUF_HEADER */
        p_buf_hdr++;

        /* type casting */
        *p_buffer = (p_BYTE) (p_buf_hdr);

        return (S_OK);
}
```

```
/************
** INCLUDE
**     Buffer Management Include File
** ----------
** FILE NAME
**     ISTABUF.H
** ----------
** DESCRIPTION
**
**     This file contains type definitions for Buffer Header and
**     Queue Header.
**
** ----------
** REVISION HISTORY
**
**   1. 03/09/92   Enghe Chimood
**        Created original
**
** ----------
** NOTES
**     Pre-include files:
**
**        SYSTEM.H
************/

/*
 *   BUFFER HEADER Type
 */

typedef struct buf_head
{              /* pointer to the next buffer  */
   struct buf_head  *p_next;

            /* Actual data byte count in the buffer */
   WORD       data_count;

}  BUF_HEADER, *p_BUF_HEADER;

#define BUF_HEADER_SIZE    sizeof(BUF_HEADER)

/*
 *   QUEUE HEADER Type
 */

typedef struct
{              /* pointer to the first buffer in queue */
   p_BUF_HEADER    p_next;

               /* pointer to the last buffer in queue */
   p_BUF_HEADER    p_last;

            /* numer of buffers in queue */
   WORD       queue_depth;

}  QHEAD,  *p_QHEAD;

#define QHEAD_SIZE    sizeof(QHEAD)
```

```
typedef struct {
        BUF_HEADER      buf_hdr;
        BYTE            addr;           /* HDLC address field */
        BYTE            cntl;           /* HDLC control field */
        BYTE            u_data[216];    /* U chip superframe data */
        BYTE            m_data[6];      /* M channel data    */
        WORD            crc_w;          /* 2 bytes CRC                */
} ISTA_FRAME;

#define FRAME_SIZE      sizeof(ISTA_FRAME)
#define BUF_COUNT       100
```

```
/*********************************************************
 * File:      ISTACNTL.C
 *
 * Description:
 *     This file contains the main loop of the ISTA module.
 *
 * Routines:
 *     issue_cmd
 *
 * Author:
 *     Enghe Chimood
 *********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"
#include "istamcro.h"
#include "hdlc.h"

Ista_core ()
{
        while (TRUE)
        {
                Head_idx = Tail_idx = 0;

                U_RESET;
                Icb->cur_state = RESET_ST;

                Ista_init ();

                U_UNRESET;
                if (NT_MODE)
                {
                        Icb->local_mode = NT;
                        Nt_init();
                }
                else
                {
                        Icb->local_mode = LT;
                        Lt_init();
                }

                Icb.cur_state = READY_ST;

                /* We are ready to operate now */
                while (TRUE)
                {
                        /* check LT/NT switch change */
                        if (mode_changed())
                                /* RESTART AGAIN */
                                break;

                        intr_evt_chk();
                }

        }
}
```

```
intr_evt_chk ()
{
INTR_EVENT event;

        while (Tail_idx != Head_idx)
        {
                event = Evt[Tail_idx];
                Evt[Tail_idx].lw = 0x0L;
                Tail_idx = (++Tail_idx) % MAX_EVT_ENTRY;

                switch (event.u.scc.port)
                {
                        case 0 :
                                Ui_pro (event.u.scc.evt);
                                break;
                        case 1:
                                Sat_pro (event.u.scc.evt);
                                break;
                        default:
                                U_chip_int (event);
                        default:
                }
        }
}

BOOLEAN mode_changed ()
{
        if (NT_MODE)
        {
                if (Icb->local_mode == LT)
                        return (TRUE);
        }
        else
                if (Icb->local_mode == NT)
                        return (TRUE);
        return (FALSE);
}

Sat_pro (event)
WORD event;
{
        /* process SCC1, HDLC event */
        switch (event)
        {
                case HDLC_RXFR:
                        hdlc_rxfr ();
                        break;

                case HDLC_TXE:
                        hdlc_txerr ();
                        break;

                case HDLC_BUSY:
                        hdlc_busy ();
                        break;

                case HDLC_TXBD:
```

```
                        hdlc_txreq ();
                        break;

                default:
                        break;
        }
}

Ui_pro (event)
WORD event;
{
        /* process SCC0, superframe rx and tx */
        switch (event)
        {
                case TP_RXBD:
                        tp_rxfr ();
                        break;

                case TP_TXE:
                        tp_txerr ();
                        break;

                case TP_BUSY:
                        tp_busy ();
                        break;

                case TP_TXBD:
                        tp_txreq ();
                        break;

                default:
                        break;
        }
}

U_chip_int (event)
INTR_EVENT event;
{
        /* process U chip interrupt */
        switch (event.u.ui.evt)
        {
                case U_IRQ3:   /* NR1 state changed */
                        U_irq3 ((BYTE) event.u.ui.u_reg);
                        break;

                case U_IRQ2:   /* eoc, R6 updated */
                        U_irq2 (event.u.ui.u_reg);
                        break;

                case U_IRQ1:   /* M4, BR1 updated */
                        U_irq1 ((BYTE) event.u.ui.u_reg);
                        break;

                case U_IRQ0:   /* M56, BR3 updated */
                        U_irq0 ((BYTE) event.u.ui.u_reg);
                        break;

                default:
```

```
                        break;
        }
}

/*********************************************************
 * routine:    issue_cmd
 *
 * description:
 *      Write the command to the IMP's command register
 *
 * arguments:
 *      cmd             the required command
 *
 * return code:
 *      0               on success
 *      error code otherwise
 *
 * side effects:
 *
 *********************************************************/
issue_cmd(cmd)
unsigned short cmd;
{
        register int i;
        register unsigned char *cr;

        cr = &(Imp->cr);

        /*
         * wait for semaphore bit
         */
        for(i = 0; i < CR_WAIT; i++)
                if( !((*cr) & CMD_FLAG) ){
                        *cr = (cmd | CMD_FLAG);
                        return(0);
                }
        return(SEMAPHORE_STUCK);
}
```

```
#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"
#include "istabuf.h"

DRIVER_CONFIG Dr {

        0x00,    /* base address in ROM */
        0xB00,   /* address of internal RAM */

        /* configuration control */
        0x700,   /* base address register */
        0x800,   /* system control register */

        /* dma */
        0x0,     /* DMA mode register */
        0x0,     /* DMA function code */

        /* interrupt controller */
        0x87a0,  /* global intr mode */
                        /* Dedicated Mode, 1,6,7 edge trigger, V7-V5 = 5 */
        0x3771,  /* interrupt mask reg */

        /* parallel port A */
        0x000f,  /* port A control */
                        /* PA0-PA3 RS449 in/out */
        0x0182,  /* port A data direction */
                        /* PA9 NT/LT input, PA8 U chip reset output,
                         * PA0-3 RS449 in/out
                         */
        0x0008,  /* port B control */
                        /* PB3 as T1N1 for IDL clock input */
        0x0007,  /* port B data direction */
                        /* PB0 Usync LED output, PB1 Satellite sync output
                         * PB2 additional LED output
                         */
        /* chip selects */
        0x0,     /* base address 0 */
        0x0,     /* option register 0 */
        0x0,     /* base address 1 */
        0x0,     /* option register 1 */
        0x0,     /* base address 2 */
        0x0,     /* option register 2 */
        0x0,     /* base address 3 */
        0x0,     /* option register 3 */

        /* timer */
        0x0,     /* timer 1 mode register */
        0x0,     /* timer 1 reference register */
        0x0,     /* timer 2 mode register */
        0x0,     /* timer 2 reference register */
        0x0,     /* watch dog timer reference resist */

        /* SCC 1 - config parameter, Transparent Mode */
        0xffff,  /* id for loopback (-1 if not) */
        0x01,    /* upper module */
        0x08,    /* number of tx buffers */
```

```
        0x08,     /* number of rx buffers */
        0x08,     /* min buffers in pool */
        0x08,     /* # of buffers in pool */
        0x0,      /* receive error mask */
        0x014,    /* event tracing */

        /* SCC common protocol parameters */
        0x0,      /* rx BD FC */
        0x0,      /* tx BD FC */
        0x0e4,    /* rx buffer length  (228 bytes) */

        0x401e,   /* SCC configuration register */
                  /* EXTC, DIV 16 clock */
        0x6003,   /* SCC mode register */
                  /* EXSYN, NTSYN, transparent MODE */
        0x00,     /* SCC sync register */
        0x17,     /* SCC interrupt mask register */
                  /* no receive char interrupt */

        /*
         * protocol specific parameter ram
         */
        0x10a,    /* scc max FRAME LENGTH */
        0x0,      /* address max */
        0x0,      /* r_addr1 */
        0x0,      /* r_addr2 */
        0x0,      /* r_addr3 */
        0x0,      /* r_addr4 */
        0xf0b8,   /* CRC constant (low word) */
        0x0,      /* CRC constant (high word) */
/* SCC 1 config end */
        /* SCC 2 - config parameter */
        0xffff,   /* id for loopback (-1 if not) */
        0x01,     /* upper module */
        0x08,     /* number of tx buffers */
        0x08,     /* number of rx buffers */
        0x08,     /* min buffers in pool */
        0x08,     /* # of buffers in pool */
        0x0,      /* receive error mask */
        0x014,    /* event tracing */

        /* SCC common protocol parameters */
        0x0,      /* rx BD FC */
        0x0,      /* tx BD FC */
        0x0e4,    /* rx buffer length (228 bytes) */

        0x401e,   /* SCC configuration register */
        0x00c0,   /* SCC mode register */
                  /* HDLC, NRZI, RTS always assert, send flag/idle */
        0x7e7e,   /* SCC sync register */
        0x1f,     /* SCC interrupt mask register */

        /*
         * protocol specific parameter ram
         */
        0x10a,    /* scc max FRAME LENGTH */
        0x0,      /* address max */
        0x0,      /* r_addr1 */
```

```
        0x0,    /* r_addr2 */
        0x0,    /* r_addr3 */
        0x0,    /* r_addr4 */
        0xf0b8,   /* CRC constant (low word) */
        0x0,    /* CRC constant (high word) */

/* SCC 2 config end */
        /* **** SCC 3 - config parameter **** */
        0xffff,   /* id for loopback (-1 if not) */
        0x01,    /* upper module */
        0x04,    /* number of tx buffers */
        0x08,    /* number of rx buffers */
        0x08,    /* min buffers in pool */
        0x08,    /* # of buffers in pool */
        0x0,    /* receive error mask */
        0x014,    /* event tracing */

        /* SCC common protocol parameters */
        0x0,    /* rx BD FC */
        0x0,    /* tx BD FC */
        0x0e4,    /* rx buffer length (228 byte) */

        0x401e,   /* SCC configuration register */
        0x1c,    /* SCC mode register */
        0x7e7e,   /* SCC sync register */
        0x1e,    /* SCC interrupt mask register */

        /*
         * protocol specific parameter ram
         */
        0x10a,    /* scc max FRAME LENGTH */
        0x0,    /* address max */
        0x0,    /* r_addr1 */
        0x0,    /* r_addr2 */
        0x0,    /* r_addr3 */
        0x0,    /* r_addr4 */
        0xf0b8,   /* CRC constant (low word) */
        0x0,    /* CRC constant (high word) */
/* **** SCC 3 config end **** */

        0x0700,   /* SCP/SMI mode register */
                /* SCP clock div 3, scp enabled, SMI disabled */

        /* physical configuration */
        0xffff,   /* serial interface mask reg */
        0x015e    /* serial interface mode reg SIMODE */
                /* B1,B2,D route to SCC1, SCC2 & SCC3 not connect
                 * to multiplexed serial interface, IDL mode supported
                 */
};

p_IMP Imp = Dr->internal_ram;
P_SCP_SMI_PRAM p_Spram;    /* SCP parameter ram addr */
P_SCP_SMI_MODE p_Smode,    /* SIMODE register address */
WORK_AREA Wa[2];          /* SCC1 and SCC2 work area */
QHEAD  Free_q;
BYTE   Buffer_area[FRAME_SIZE * BUF_COUNT];
```

```
INTR_EVENT Evt[MAX_EVT_ENTRY];      /* interrupt event entry table
WORD Head_idx, Tail_idx;
```

```
/*********************************************************
* File:   ISTAINIT.C
*
* Description:
*  Contains initialization routine for ISTA
*
* Routines:
*  Ista_init
*  imp_configure
*
* Author:
*  Enghe Chimood
*********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"

extern int scc0_intr()
          , scc1_intr()
          , tim0_intr()
          , scc2_intr()
          , tim1_intr()
          , scp_intr()
          , tim2_intr()
          , gpip0_intr()
          , error_intr();

/*********************************************************
 * routine: ista_init
 *
 * description:
 *  Initialize the whole ISTA
 *
 * arguments:
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/

ista_init()
{
BYTE i;

          /* create buffers for operation */
          Ista_init_bufmnt();

          /*
           * write IMP configuration registers
           * and issue the software-reset command
           */
          imp_configure();

          /*
           * initialize SCC1 and SCC2, BUT NOT SCC3
```

```
        */
        for(i = 0; i < 2; i++)
        {
                init_wa (i);
                scc_init(i, &Dr->scc[i])
        }

        /* initialize SCC for specific operation
         * SCC1 -> transparent operation for IDL interface
         * SCC2 -> HDLC operation of satellite interface
         * SCC3 -> for SCP interface
         *
         * Transparent Mode was done in scc_init already
         */
        hdlc_init(1, Dr->scc[1]);

        timer_init();
}

/*********************************************************
 * routine: scc_init
 *
 * description:
 * Initialize the global SCC parameters.
 * These are parameters of relevance to
 * all SCC's, irrespective of their mode
 * of operation.
 *
 * arguments:
 * port   the number of the SCC port (0, 1) = SCC1, SCC2
 * scc    points to the SCC configuration structure in Dr
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
scc_init(port, scc)
struct scc *scc;
{
struct scc_pram *spram;
struct scc_regs *sregs;

        spram = &Imp->pram[port].scc;
        sregs = &Imp->scc_regs[port];

        /*
         * first "empty" the 68302 Tx and Rx BD tables
         */
        spram->rbd[0].status = 0;
        spram->tbd[0].status = 0;

        /*
         * set internal registers
         */
        Imp->scc_regs[port].conf = scc->conf;
        Imp->scc_regs[port].sync = scc->sync;
        Imp->scc_regs[port].mask = scc->mask;
```

```
        Imp->scc_regs[port].mode = scc->mode;

        /*
         * set scc common protocol parameters
         * SAME FOR BOTH TXPARENT AND HDLC MODE
         *  rx BD FC
         *  tx BD FC
         *  maximum rx buffer len
         */
        spram->pscc.h.rfc = scc->rfc;
        spram->pscc.h.tfc = scc->tfc;
        spram->pscc.h.rblen = scc->rlen;

        /*
         * prepare the transmission table
         */
        scc_itx(port, scc->tsize);

        /*
         * prepare the receive table
         */
        scc_irx(port, scc->rsize);
        fill_rtab(port);
}


/*********************************************************
 * routine: scc_irx
 *
 * description:
 *  Intialize the SCC receive BD table.
 *  All status bits (except Wrap on the last BD)
 *  are cleared (the Empty bit will be set later
 *  for each BD by fill_rtab).
 *
 * arguments:
 *  port      the SCC port involved
 *  rsize     the number of BD's in the table
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
scc_irx(port, rsize)
BYTE port;
BYTE rsize;
{
BYTE i;

        for(i = 0; i < rsize; i++){
                (Imp->pram[port].scc.rbd + i)->status = 0;
        }
        (Imp->pram[port].scc.rbd + (rsize-1))->status = R_W;
}

/*********************************************************
 * routine: scc_itx
```

```
*
* description:
*  Initialize an SCC transmit table.
*
* arguments:
*  port      the SCC port involved
*  tsize     the size of the tx table
*
* return code:
*
* side effects:
*
*********************************************************/
scc_itx(port, tsize)
BYTE port;
BYTE tsize;
{
BYTE i;
        for(i = 0; i < tsize; i++){
                (Imp->pram[port].scc.tbd + i)->status = T_X;
        }
        (Imp->pram[port].scc.tbd + (tsize-1))->status |= T_W;
}

/*********************************************************
 * routine: imp_configure
 *
 * description:
 *  Initialze the IMP by writing all of its configuration
 *  registers.
 *
 * arguments:
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
imp_configure()
{
        /*
         * write system configuration registers
         */
        WRITE_BAR(Dr->bar);
        WRITE_SCR(Dr->scr);

        /*
         * issue software reset command
         */
        issue_cmd(SOFTWARE_RESET);

        /* dma */
        Imp->dma_mode = Dr->dma_mode;
        Imp->dma_fc = Dr->dma_fc;

        /* interrupt controller */
        Imp->intr_mode = Dr->intr_mode;
        Imp->intr_mask = Dr->intr_mask;
```

```
        /* parallel port A */
        Imp->pa_control = Dr->pa_control;
        Imp->pa_direction = Dr->pa_direction;
        Imp->pb_control = Dr->pb_control;
        Imp->pb_direction = Dr->pb_direction;

        /* chip selects */
        Imp->cs0_base = Dr->cs0_base;
        Imp->cs0_option = Dr->cs0_option;
        Imp->cs1_base = Dr->cs1_base;
        Imp->cs1_option = Dr->cs1_option;
        Imp->cs2_base = Dr->cs2_base;
        Imp->cs2_option = Dr->cs2_option;
        Imp->cs3_base = Dr->cs3_base;
        Imp->cs3_option = Dr->cs3_option;

        /* timers */
        Imp->t0_mode = Dr->t0_mode;
        Imp->t0_ref = Dr->t0_ref;
        Imp->t1_mode = Dr->t1_mode;
        Imp->t1_ref = Dr->t1_ref;
        Imp->wd_ref = Dr->wd_ref;

        /* SCP port MODE */
        Imp->sp_scm = Dr->scp_smi_mode;
        p_Spram = (P_SCP_SMI_PRAM) &Imp->pram[2].scc.tbd[4];
        p_Smode = (P_SCP_SMI_MODE) &Imp->sp_scm;

        /* serial interface */
        Imp->si_mask = Dr->si_mask;
        Imp->si_mode = Dr->si_mode;
}

init_wa (port)
BYTE port;
{

}

/*******************************************************
 * routine: hdlc_init
 *
 * description:
 *  Initialize the driver tables for
 *  HDLC ports.
 *
 * arguments:
 *  port      port #
 *  hdlc      pointer of scc configuration structure
 *
 * return code:
 *
 * side effects:
 *
 *******************************************************/
hdlc_init(port, scc)
struct scc *scc;
```

```
{
struct scc_pram *spram;

        spram = &Imp->pram[port].scc;
        /*
         * initialize hdlc parameter ram
         */
        spram->pscc.h.r_max = scc->pram.h.r_max;
        spram->pscc.h.r_mask = scc->pram.h.r_mask;
        spram->pscc.h.r_addr1 = scc->pram.h.r_addr1;
        spram->pscc.h.r_addr2 = scc->pram.h.r_addr2;
        spram->pscc.h.r_addr3 = scc->pram.h.r_addr3;
        spram->pscc.h.r_addr4 = scc->pram.h.r_addr4;
        spram->pscc.h.r_cmaskl = scc->pram.h.r_cmaskl;
        spram->pscc.h.r_cmaskh = scc->pram.h.r_cmaskh;

        /*
         * zero counters
         */
        spram->pscc.h.discard_cnt = 0;
        spram->pscc.h.crc_cnt = 0;
        spram->pscc.h.abort_cnt = 0;
        spram->pscc.h.nmf_cnt = 0;
        spram->pscc.h.retx_cnt = 0;
}
```

```
#define    SCP_ENABLE  (Imp->pa_data = Imp->pa_data & (~ SCP_EN_MSK))

#define    SCP_DISABLE (Imp->pa_data = Imp->pa_data | SCP_EN_MSK)

#define    U_UNRESET    (Imp->pa_data = Imp->pa_data | U_RESET_MSK)

#define    U_RESET    (Imp->pa_data = Imp->pa_data & (~U_RESET_MSK))

#define    LED_ON(a)   (Imp->pb_data |= a)

#define    LED_OFF(a)  (Imp->pb_data ^= a)
```

```
/*****************************************************
 * File:     ISTAUPRO.C
 *
 * Description:
 *     This file contains routines that process the
 *     U chip related event and frames.
 *
 * Routines:
 *
 *
 *
 * Author:
 *     Enghe Chimood
 *
 *****************************************************/
U_irq3 (nr1)
BYTE nr1;
{

}

U_irq2 (nr6)
WORD nr6;
{


}

U_irq1 (br1)
BYTE br1;
{

}

U_irq0 (br3)
BYTE br3
{

}
```

```
/*
 * registers.h
 * Definitions of the IMP registers
 */


/*****************************************************************
        Base Address Register (address 0xf2)
*****************************************************************/

#define WRITE_BAR(x)    (*((short *)0xf2) = x)

typedef struct bar_reg {
        BIT fc  : 3;    /* 3 bit function code */
        BIT cfc : 1;    /* compare fc */
        BIT base : 12;  /* internal ram base address (bits 12-23) */
}BAR_REG;


/*****************************************************************
        System Control Register (address 0xf4)
*****************************************************************/

typedef struct scr_reg {
        BIT : 4;
        BIT ipa : 1;    /* interrupt priority active */
        BIT hwt : 1;    /* h/w watch dog timer */
        BIT wpv : 1;    /* write protect violation */
        BIT adc : 1;    /* address decode conflict */
        BIT : 3;
        BIT wpve : 1;   /* write protect violation enable */
        BIT rmcst : 1;  /* RMC cycles special treatment */
        BIT emws : 1;   /* external master wait state */
        BIT adce : 1;   /* address decode conflict enable */
        BIT bclm : 1;   /* bus clear mask */
        BIT frzw : 1;   /* freeze watchdog timer enable */
        BIT frz2 : 1;   /* freeze timer2 enable */
        BIT frz1 : 1;   /* freeze timer1 enable */
        BIT : 1;
        BIT hwden : 1;  /* hardware watchdog enable */
        BIT hwdcn : 3;  /* hardware watchdog count */
        BIT rsten : 1;  /* reset enable */
        BIT div16 : 1;  /* low power prescale divide by 16 */
        BIT lpen : 1;   /* low power enable */
        BIT lpdiv : 5;  /* low power clock divider */
}SCR_REG;

#define WRITE_SCR(x)    (*((long *)0xf4) = x)
#define CLEAR_IPA   (((struct scr_reg *)0xf4)->ipa = 1)


/*****************************************************************
        Command Register
*****************************************************************/

/* bit fields within command register */
#define SOFTWARE_RESET  0x80
#define CMD_GCI     0x40
```

```
#define CMD_OPCODE  0x30
#define CMD_CHANNEL 0x06
#define CMD_FLAG    0x01

/* command opcodes */
#define STOP_TX     0x00
#define RESTART_TX  0x10
#define ENTER_HUNT_MODE 0x20
#define RESET_RX_BCS    0x30    /* bisync only */
#define GCI_ABORT   CMD_GCI | 0x04
#define GCI_TIMEOUT CMD_GCI | 0x14



/*****************************************************************
        SCC configuration register
*****************************************************************/

typedef struct scc_conf {
        BIT woms : 1;   /* wire-ORed mode */
        BIT extc : 1;   /* external clock */
        BIT txclk: 1;   /* tx clock source */
        BIT rxclk: 1;   /* rx clock source */
        BIT clock: 11;  /* clock divider */
        BIT div4 : 1;   /* prescaler divide by 4 */
}SCC_CONF;



/*****************************************************************
        SCC mode register
*****************************************************************/

/* SCC modes */
#define HDLC_PORT     0
#define ASYNC_PORT    1  /* uart or ddcmp */
#define DDCMP_PORT    2
#define BISYNC_PORT   3

/*
 * diagnostics bits
 */
#define NORMAL    0  /* normal operation with CTS and CD */
                        /* automatically controlled by SCC */
#define LOOPBACK  1  /* connect tx to rx */
#define SERIAL_ECHO 2  /* connect rx to tx */
#define NORMAL_SW  3  /* normal operation with CTS and CD */
                        /* under control of s/w */


/* hdlc mode register */

typedef struct hdlc_mode_reg {
        BIT min_flags : 4;  /* min flags between frames */
        BIT crc_32 : 1; /* 32-bit CRC */
        BIT : 2;
        BIT retx : 1;   /* enable retx */
        BIT flg : 1;    /* send flags between frames */
```

```
        BIT data_enc : 1;  /* data encoding (NRZ/NRZ1) */

        BIT diag : 2;      /* diagnostic mode */
        BIT rx_enable : 1;     /* rx enable */
        BIT tx_enable : 1;     /* tx enable */
        BIT mode : 2;
}HDLC_MODE_REG;

/*****************************************************************
        SCP/SMI mode register
*****************************************************************/

/*
 * defines for SMI modes
 */
#define IOM_UNUSED  0  /* IOM: monitor not used */
#define IOM_DCHAN   1  /* IOM: monitor controls D channel */
#define IOM_DATA_CNTRL  2  /* IOM: monitor txs data/control bytes */
#define IDL_AM_HUNT 4  /* IDL: hunt M and A channels on zero */
#define IDL_M_HUNT  5  /* IDL: hunt M channel on zero */
#define IDL_A_HUNT  6  /* IDL: hunt A channel on zero */
#define IDL_NORMAL  7  /* IDL: normal (no hunt on zero mode) */

typedef struct scp_smi_mode {
/* SCP bits */
        BIT scp_start : 1;     /* start SCP tx bit */
        BIT scp_loop : 1;      /* SCP loopback mode */
        BIT : 1;
        BIT scp_prescale : 4;  /* SCP prescale modulus select */
        BIT scp_enable : 1;    /* enable SCP */
/* SMI bits */
        BIT : 2;
        BIT smi_mode : 3;      /* SMI mode */
        BIT smi_loop : 1;      /* SMI loopback mode */
        BIT smi2_enable : 1;   /* enable SMI2 */
        BIT smi1_enable : 1;   /* enable SMI1 */
} SCP_SMI_MODE, P_SCP_SMI_MODE;



/*****************************************************************
        serial interface mode register
*****************************************************************/

/*
 * defines for D/B channel routing
 * (b2_rout, b1_rout, d_rout)
 */
#define NOT_SUPPORTED  0
#define ROUTE_SCC0  1
#define ROUTE_SCC1  2
#define ROUTE_SCC2  3

/*
 * serial interface mode
 */
#define IDL_MODE   0
#define IOM_MODE   1
```

```
#define PCM_MODE   2
#define NMSI_MODE  3

typedef struct si_mode {
        BIT stz : 1;   /* set L1TxD to 0 (IOM) */
        BIT sync_8 : 1; /* PCM only */
        BIT loopback : 1;  /* loopback mode */
        BIT echo : 1;  /* echo mode */
        BIT : 2;
        BIT b2_rout : 2;   /* B2 routing (IDL, IOM) */
        BIT b1_rout : 2;   /* B1 routing (IDL, IOM) */
        BIT d_rout : 2; /* D routing (IDL, IOM) */
        BIT nmsi_3 : 1; /* SCC3 routing */
        BIT nmsi_2 : 1; /* SCC2 routing */
        BIT mode : 2;
} SI_MODE;


/*****************************************************************
        serial interface mask register
*****************************************************************/

typedef struct si_mask {
        BIT b2 : 8;
        BIT b1 : 8;
} SI_MASK;



/*****************************************************************
        interrupt registers
*****************************************************************/

#define INTR_PB11  0x8000    /* parallel port B bit 11 */
#define INTR_PB10  0x4000    /* parallel port B bit 10 */
#define INTR_SCC0  0x2000    /* SCC port 0 */
#define INTR_DMAERR 0x1000    /* dma error */
#define INTR_DMA   0x0800    /* dma */
#define INTR_SCC1  0x0400    /* SCC port 1 */
#define INTR_TIMER0 0x0200    /* timer 0 */
#define INTR_SCC2  0x0100    /* SCC port 2 */
#define INTR_PB9   0x0080    /* parallel port B bit 9 */
#define INTR_TIMER1 0x0040    /* timer 1 */
#define INTR_SCP   0x0020    /* SCP port */
#define INTR_TIMER2 0x0010    /* timer 2 */
#define INTR_SMI0  0x0008    /* SMI port 0 */
#define INTR_SMI1  0x0004    /* SMI port 1 */
#define INTR_PB8   0x0002    /* parallel port B bit 8 */
#define INTR_ERR   0x0001    /* error */

typedef struct intr_reg {
        BIT pb11 : 1;  /* parallel port B bit 11 */
        BIT pb10 : 1;  /* parallel port B bit 10 */
        BIT scc0 : 1;  /* SCC port 0 */
        BIT dmaerr : 1; /* dma error */
        BIT dma : 1;   /* dma */
        BIT scc1 : 1;  /* SCC port 1 */
        BIT timer1 : 1; /* timer 1 */
```

```
        BIT scc2 : 1;   /* SCC port 2 */
        BIT pb9 : 1;    /* parallel port B bit 9 */
        BIT timer2 : 1; /* timer 2 */
        BIT scp : 1;    /* SCP port */
        BIT timer3 : 1; /* timer 3 */
        BIT smi0 : 1;   /* SMI port 0 */
        BIT smi1 : 1;   /* SMI port 1 */
        BIT pb8 : 1;    /* parallel port B bit 8 */
        BIT error : 1;  /* error */
} INTR_REG;



/*****************************************************************
        dma mode register
*****************************************************************/

/*
 * request generation
 */
#define EXTERNAL_REQG(x)   (x & 2)
#define INTREQ_LIM     0
#define INTREQ_MAX     1
#define EXTREQ_BURST       2
#define EXTREQ_CYCL    3

typedef struct dma_mode_reg {
        BIT : 1;        /* reserved */
        BIT ext_cntl : 1;   /* external control option */
        BIT intr : 1;   /* interrupt on completion */
        BIT intr_e : 1; /* interrupt on error */
        BIT req_gen : 2;    /* request generation options */
        BIT isource : 1;    /* increment source address */
        BIT idest : 1;  /* increment dest address */
        BIT ssize : 2;  /* source size */
        BIT dsize : 2;  /* destination size */
        BIT burst : 2;  /* burst transfer control */
        BIT reset : 1;  /* software reset */
        BIT start : 1;  /* start bit */
} DMA_MODE_REG;


/*****************************************************************
        chip select option register
*****************************************************************/
#define DTACK      0xe000
#define ADR_MASK   0x1ffc
#define RW_MASK    0x0002
#define FC_MASK    0x0001

/*****************************************************************
        U interface chip registers
*****************************************************************/

/* U interface read/write bit */
#define    U_WRITE    0x00
#define    U_READ     0x80
```

```
/* nibble register addr/id */
#define   NR0 0x00
#define   NR1 0x10
#define   NR2 0x20
#define   NR3 0x30
#define   NR4 0x40
#define   NR5 0x50

/* NR6 uses 16-bit access */
#define   NR6 0x6000

#define   BR0 0x70
#define   BR1 0x71
#define   BR2 0x72
#define   BR3 0x73
#define   BR4 0x74
#define   BR5 0x75
#define   BR6 0x76
#define   BR7 0x77
#define   BR8 0x78
#define   BR9 0x79
#define   BR10   0x7a
#define   BR11   0x7b
#define   BR12   0x7c
#define   BR13   0x7d
#define   BR14   0x7e
#define   BR15   0x7f

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT reset  : 1;   /* software reset */
        BIT pd_en  : 1;   /* power down enable */
        BIT pdown  : 1;   /* absolute power down */
        BIT normal : 1;   /* return to normal */
} T_NR0;

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT linkup : 1;   /* link up */
        BIT error  : 1;   /* error indication */
        BIT sf_sync : 1;   /* superframe sync */
        BIT trans_act_prog; :1; /* transparent/activation in progress */
} T_NR1;

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT act_req : 1;   /* activation request */
        BIT deact_req  : 1; /* deactivation request */
        BIT sf_up_dis  : 1; /* superframe update disable */
        BIT cus_en     : 1; /* customer enable */
} T_NR2;
```

```
typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT nr1_chg : 1;   /* IRQ3 - state change in NR1 */
        BIT eoc_chg : 1;   /* IRQ2 - eoc NR6 is updated */
        BIT m4_chg  : 1;   /* IRQ1 - m4 buf BR1 is updated */
        BIT m56_chg : 1;   /* IRQ0 - m50, m51, m60 bits updated */
} T_NR3;

#define U_IRQ3  0x08
#define U_IRQ2  0x04
#define U_IRQ1  0x02
#define U_IRQ0  0x01

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT irq3_en : 1;   /* enable IRQ3 */
        BIT irq2_en : 1;   /* enable IRQ2 */
        BIT irq1_en : 1;   /* enable IRQ1 */
        BIT irq0_en : 1;   /* enable IRQ0 */
} T_NR4;

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT res : 1;   /* reserved */
        BIT b1  : 1;   /* block B1 */
        BIT b2  : 1;   /* block B2 */
        BIT b1_b2_s : 1;   /* swap B1/B2 */
} T_NR5;

typedef struct
{
        BIT rw  : 1;   /* read or write */
        BIT addr   : 3;   /* address */
        BIT eoc : 12;  /* eoc bits */
} T_NR6;

typedef struct
{
        union
        {
        struct
        {
                BIT    act : 1;   /* act bit */
                BIT    dea : 1;   /* deact bit */
                BIT    res : 6;   /* reserved */
        } lt;
        struct
        {
                BIT    act : 1;   /* act bit */
                BIT    ps1 : 1;   /* power status bit */
                BIT    ps2 : 1;   /* power status bit */
                BIT    ntm : 1;   /* NT in test mode */
```

```
                BIT    cso : 1;   /* NT cold start only */
                BIT    res : 3;   /* reserved */
        } nt;
        BYTE   m4_byte;
        } u1;
} T_BR0_BR1;

typedef struct
{
        BIT res    : 3; /* M50, M60, M51 bits */
        BIT febe_input  : 1; /* febe input */
        BIT res1       : 4;
} T_BR2;

typedef struct
{
        BIT dummy      : 3; /* M50, M60, M51 bits */
        BIT rx_febe    : 1; /* received febe */
        BIT cmp_nebe   : 1; /* computed CRC check */
        BIT ver_act    : 1; /* verified act */
        BIT ver_dea    : 1; /* verified deact */
        BIT sf_detect  : 1; /* superframe detect */
} T_BR3;


/* BR4 contains febe counter, BR5 contains nebe counter */

typedef struct
{
        BIT b1_loop    : 1; /* B1 loop */
        BIT b2_loop    : 1;
        BIT bd_loop    : 1;
        BIT trpt_loop  : 1;
        BIT b1_idl_loop : 1;
        BIT b2_idl_loop : 1;
        BIT bd_idl_loop : 1;
        BIT trpt_idl_loop  : 1;
} T_BR6;

typedef struct
{
        BIT res    : 3;   /* reserved */
        BIT idl_invert  : 1;   /* IDL invert */
        BIT idl_free_run   : 1;   /* IDL free run */
        BIT idl_speed   : 1;   /* IDL clock speed */
                                                /*   0 = 2.56 MHz, 1 = 2.048 MHz */
        BIT idl_ms_invert   : 1;   /* IDL master or slave invert */
        BIT idl_bits_mode   : 1;   /* 1 = 8 bit mode, 0 = 10 bit mode */
} T_BR7;

typedef struct
{
        BIT
        BIT
        BIT
} T_BR8;

typedef struct
```

```
{
        BIT eoc_cntl    : 2;    /* eoc control */
        BIT m4_cntl     : 2;    /* M4 control */
        BIT m5m6_cntl   : 2;    /* M5/M6 control */
        BIT febe_nebe   : 1;    /* febe/nebe control */
        BIT res     : 1;    /* reserved */
} T_BR9;

/* BR10 is reserved byte */

typedef struct
{
        BIT act_cntl    : 7;    /* activation control */
        BIT act_timer_dis   : 1;    /* activation timer disable */
} T_BR11_WRITE;

typedef struct
{
        BIT act_state   : 7;    /* activation state */
        BIT act_timeout : 1;    /* activation timer expire */
} T_BR11_READ;

/* BR12, BR13 NOT defined */

typedef struct
{
        BIT res     : 1; /* reserved */
        BIT rw_cntl     : 1; /* 1 = ro/wo become rw */
        BIT res1        : 1; /* reserved */
        BIT fr_defr_loop    : 1; /* framer to deframer loop */
        BIT tone_cntl   : 1; /* superframe tone control */
        BIT res1        : 2; /* reserved */
        BIT clk_en      : 1; /* 1 = enable the clock */
} T_BR14;


/* BR15 is only used to read revision number  */@
```

```
/*********************************************************
 * File:    scc.c
 *
 * Description:
 *  This file contains routines used by SCC controllers
 *  to initialize an SCC channel
 *  irrespective of their mode of operation.
 *
 * Routines:
 *  scc_pool
 *  scc_irx
 *  scc_itx
 *
 * Author:
 *  Enghe Chimood
 *********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"

/*********************************************************
 * routine: scc_pool
 *
 * description:
 *  Build a pool of receive buffers.
 *  Any buffers currently allocated to the receive
 *  pool of this port are released, new buffers
 *  will be allocated.
 *  Note that an extra R_FRAME is allocated at the
 *  end of the pool. This guarantees that the linked list
 *  of R_FRAMES comprising the rx-queue/rx-pool is
 *  never empty (an R_FRAME in the linked list is never
 *  linked to the receive BD table if its nextb pointer
 *  is null).
 *
 * arguments:
 *  port       the scc port involved (0, 1 or 2)
 *  num     number of buffers (not including extra
 *          R_FRAME at end of pool)
 *  len     the buffers length
 *
 * return code:
 *  0       on success
 *  non-zero if not enough buffers were available
 *
 * side effects:
 *
 *********************************************************/
scc_pool(port, num, len)
{
        int i,
/*
        wa->imp_rbd = wa->pram->rbd;
        wa->imp_pbd = wa->pram->rbd;
        wa->rfree = wa->rsize;
```

```
*/

        /*
         * prepare the receive table
         */
        scc_irx(port, wa->rsize);
        fill_rtab(wa);
        return(i<num);
}


/*********************************************************
 * routine: scc_irx
 *
 * description:
 *  Intialize the SCC receive BD table.
 *  All status bits (except Wrap on the last BD)
 *  are cleared (the Empty bit will be set later
 *  for each BD by fill_rtab).
 *
 * arguments:
 *  port      the SCC port involved
 *  rsize     the number of BD's in the table
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
scc_irx(port, rsize)
{
        int i;
/*
        wa->rfree = wa->rsize;
        wa->imp_rbd = wa->pram->rbd;
        wa->imp_pbd = wa->pram->rbd;
        wa->rcount = 0;
        wa->rerr_cnt = 0;
*/

        for(i = 0; i < rsize; i++){
                (Imp->pram[port].scc.rbd + i)->status = 0;
        }
        (Imp->pram[port].scc.rbd + (rsize-1))->status = R_W;
}



/*********************************************************
 * routine: scc_itx
 *
 * description:
 *  Initialize an SCC transmit table.
 *
 * arguments:
 *  port      the SCC port involved
 *  tsize     the size of the tx table
 *
```

```
 * return code:
 *
 * side effects:
 *
 ********************************************************/
scc_itx(port, tsize)
{
        int i;
        register IMP_BD *b;
        unsigned char *s;
/*
        wa->imp_tbd = wa->pram->tbd;
        wa->imp_cbd = wa->pram->tbd;
        wa->tcount = 0;
        wa->terr_cnt = 0;
*/
        for(i = 0; i < tsize; i++){
                (Imp->pram[port].scc.tbd + i)->status = T_X;
        }
        (Imp->pram[port].scc.tbd + (tsize-1))->status |= T_W;

}
```

```
/*******************************************************
 * File:    scc_r.c
 *
 * Description:
 *          This file contains routines used for handling
 *          data reception of SCCs
 *
 * Routines:
 *          driver_rx
 *          fill_rtab
 *          replenish_rxbd
 *          return_to_pool
 *
 * Author:
 *          Enghe Chimood
 *******************************************************/

#include "system.h"
#include "hdlc.h"
#include "imp.h"
#include "register.h"

/*******************************************************
 * routine:          fill_rtab
 *
 * description:
 *          Fill the receive buffer table with new buffers as possible.
 *          Called after collecting received frames (XXX_rxfr),
 *          when returning buffers to the receive pool (return_to_pool)
 *          and when creating a pool (scc_pool).
 *
 * arguments:
 *
 * return code:
 *          The number of buffers added to the rx table.
 *
 * side effects:
 *          Increment (modulo the receive table size) the imp_pbd
 *          index each tme a new buffer is linked to the table.
 *
 *******************************************************/
fill_rtab(port)
BYTE port;
{
          int count;
          p_BYTE p_buf;
          IMP_BD *rbd;              /* BD in table */

          rbd = Imp->pram[port].scc.rbd;
          rfree = Dr.scc[port].rsize;     /* number of rx buffer */

          count = 0;
          while( rfree ){
                    /*
                     * prepare status bits, leaving the
                     * wrap bit unaltered
                     */
                    rbd->status &= R_W;
```

```
/* EC debug
                rbd->status |= wa->BDrx;
*/
                rbd->length = 0;

                Ista_get_buf (&p_buf);

                /* for SCC1, IDL superframe starts from 2 byte offset in the
                 * data area
                 */
                if (port == 0)
                        p_buf += 2;

                rbd->buf = r->buf;

                /*
                 * advance to next frame, BD
                 */
                rfree--;
                rbd->status |= R_E;             /* ready for reception */

                count++;
                rbd++;
        }

        return(count);
}



/*********************************************************
 * routine:         replenish_rxbd
 *
 * description:
 *      Replenish the Rx BD table.
 *
 * arguments:
 *      wa                  points to the work area
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
replenish_rxbd(wa)
register WORK_AREA *wa;
{
        short flgs;
        int i;

        /*
         * link frames to BD table
         */
        i = fill_rtab(wa);
        flgs = wa->flags & (POOL_EMPTY | LOCAL_BUSY);

        /*
         * if there are less than minpool buffers
```

```
		* in the rx pool, issue a busy interrupt
		* and reduce the maximum frame length.
		* (Note that in non-HDLC modes, the
		* minpool value is set to zero).
		*/
		if( wa->pool_cnt < wa->minpool ){
			if( !(flgs & POOL_EMPTY) ){
				wa->pram->pscc.h.r_max = wa->rlen;
				wa->flags |= POOL_EMPTY;
			}
		} else if( flgs & POOL_EMPTY ){
			wa->flags &= ~POOL_EMPTY;
			wa->pram->pscc.h.r_max = wa->fr_lng;
		}

		/*
		* clear local busy flag
		* if any buffers were added to the rx BD table
		*/
		if( i )
			wa->flags &= ~LOCAL_BUSY;

		/*
		* local busy situation may have either been
		* cleared or detected
		*/
		if( flgs ){
			if( !(wa->flags & (POOL_EMPTY | LOCAL_BUSY)) )
				return(LOCAL_BUSY_CLEARED);
			else
				return(0);
		} else if( wa->flags & (POOL_EMPTY | LOCAL_BUSY) )
				return(LOCAL_BUSY_DETECTED);
			else
				return(0);
}


/*******************************************************
 * routine:		driver_rx
 *
 * description:
 *		This routine either enables, activates or
 *		disables the functioning of the receiver.
 *		Since the message is the address of a local
 *		variable, it cannot be sent.
 *
 * arguments:
 *		m			points to message received
 *
 * return code:
 *
 * side effects:
 *
 *******************************************************/
driver_rx(m)
MSG *m;
{
```

```
        GCT *gct;
        CHILD *child;
        WORK_AREA *wa;
        long rxtype;

        GETGCT(gct);
        child = gct->driver->child + m->hdr.id;
        wa = (WORK_AREA *)child->work_area;
        rxtype = (RX_IND << 16) | m->hdr.id;
        switch( m->param[0] ){
        case ENABLE:
                wa->flags &= ~RX_DISABLE;
                (*child->smac[SUB(RX_IND)])(&rxtype, child);
                break;
        case ACTIVATE:
                wa->flags |= RX_ACTIVATE;
                (*child->smac[SUB(RX_IND)])(&rxtype, child);
                break;
        case DISABLE:
                wa->flags |= RX_DISABLE;
                break;
        default:
                return(BAD_TYPE);
        }
        return(0);
}

/*********************************************************
 * routine:         return_to_pool
 *
 * description:
 *       Return the given R_FRAMES to the receive pool.
 *       All frames are automatically assumed to be from the
 *       same SCC port. Buffers of a frame are linked via nextb.
 *
 * arguments:
 *       r                  points to the first r_frame in the chain
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
return_to_pool(r, child)
register R_FRAME *r;
CHILD *child;
{
        register WORK_AREA *wa;
        register R_FRAME *r1, *r2;
        int i, status;
        MSG *m;
        GCT *gct;
        extern MSG *getm();

        GETGCT(gct);
        wa = (WORK_AREA *)child->work_area;
        r1 = r;                     /* save r in temp */
        r2 = wa->rtp_q;             /* save rtp in temp */
```

```
        /*
         * find the end of the frame
         * NOTE: be careful noy to zero the RX_POOL
         * bit which should already be set in flags.
         */
        r->flags &= ~LINKED_TO_TABLE;
        r->hdr.status = 0;
        wa->pool_cnt++;
        driver_trace(RTP, r->hdr.id, r );
        while( r->nextb ){
                wa->pool_cnt++;
                r = r->nextb;
                r->hdr.status = 0;
                r->flags &= ~LINKED_TO_TABLE;
                driver_trace(RTP, r->hdr.id, r );
        }

        /*
         * to be correct, the nextb of the old rtp
         * should only be written after the rtp has
         * been written with its new value
         */
        wa->rtp_q = r;                  /* last buffer */
        r2->nextb = r1;

        /*
         * keep receive BD table stoked up
         */
        status = replenish_rxbd(wa);
        if( status ){
                driver_trace(BUSY_IND, r->hdr.id, status);
                m = getm(BUSY_IND, r->hdr.id, 0, 0);
                if( m ){
                        m->hdr.status = status;
                        if( (*gct->send)(child->upper, m) )
                                relm(m);
                }
        }
}
```

```
/*********************************************************
 * File:   scc_t.c
 *
 * Description:
 *  This file contains routines that handle
 *  transmission of data over SCC channels.
 *  The routines here are used for all modes of
 *  operation (HDLC, UART, DDCMP and BISYNC).
 *
 * Routines:
 *  transmit_frames
 *  confirm_frame
 *  transmit_error
 *  driver_conf
 *
 * Author:
 *  Enghe Chimood
 *********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"


/*********************************************************
 * routine: transmit_frames
 *
 * description:
 *  Attach one frame to the corresponding
 *  IMP transmit BD table.
 *  On entry, it is assumed that only the WRAP bit may be
 *  set in any empty BD.
 *
 * arguments:
 *  wa      points to the work area
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
transmit_frames(wa)
register WORK_AREA *wa;
{
        register BYTE *n;
        register int i;
        register IMP_BD *bd1, *bd2;
        register LAYER_INFO *l;
        register IMP_BD *first;
        unsigned char *soh, *stx, *etx, *eot;
        IMP_BD *bd3;

        /*
         * tx_q is the next frame for transmission
         */
        n = wa->tx_q;
        first = 0;
```

```
if ( n ){

        /*
         * see if enough buffers in BD table
         */
        if( wa->tfree == 0 )
                break;
        wa->tfree -;

        bd1 = wa->imp_tbd;

        /*
         * clear status bit (except wrap bit)
         * and set length and ptr fields in BD
         */
        bd1->status &= (T_W | T_X);
        if( l->flags & BISYNC_TRANSP )
                bd1->status |= T_TR;
        bd1->buf = &(n->hbuf[l->hoff]);
        bd1->length = n->hsize - l->hoff;
        bd1->status |= T_R;

        /*
         * prepare index for data buffer
         */
        if( bd1->status & T_W )
                wa->imp_tbd = wa->pram->tbd;
        else
                wa->imp_tbd++;

        bd1->status |= (T_I | T_L);

        /*
         * set optional bits for
         * first buffer in frame
         */
        bd1->status |= wa->BDfirst;

        /*
         * set per-protocol
         * status bits in last BD
         */
        bd1->status |= wa->BDlast;

 }

 /*
  * update pointers and
  * set Ready bit in first BD
  */
 wa->tx_q = NULL;
 bd1->status |= T  R;
}


/*******************************************************
```

```
* routine: confirm_frame
*
* description:
*  This is the routine common to all modes of
*  SCC operations (HDLC, UART, DDCMP and BISYNC).
*  It is called after receiving a TX_CONF message
*  (typically, from an interrupt routine).
*
* arguments:
*  wa     points to the channel's work area
*
* return code:
*  0      no more frames to confirm
*  otherwise   pointer to next frame for confirming
*
* side effects:
*
*********************************************************/
T_FRAME *
confirm_frame(wa)
register WORK_AREA *wa;
{
        register IMP_BD *b;
        register T_FRAME *t, *t1;

        /*
         * confirm all transmitted BD's
         */
        t = wa->conf_q;
        t1 = 0;
        if( t ){
                /*
                 * if not linked to the tx table,
                 * there's no need to proceed
                 */
                if( !(t->layer[0].flags & LINKED_TO_TABLE) )
                        goto out;

                /*
                 * for each BD of t ...
                 */
                do {
                        b = wa->imp_cbd;

                        /*
                         * confirm only if transmitted:
                         * if an error was encountered,
                         * the remaining buffers will never be
                         * transmitted.
                         */
                        if( b->status & T_R )
                                goto out;
                        if( b->status & T_ERROR ){
                                if( wa->conf_q )
                                        driver_trace(TX_ERR,
                                                wa->conf_q->hdr.id, b);
                                transmit_error(wa);
                                goto out;
```

```
                        }

                        /*
                         * get next index in BD table
                         * for confirming
                         */
                        if( b->status & T_W )
                                wa->imp_cbd = wa->pram->tbd;
                        else
                                wa->imp_cbd++;

                        wa->tfree++;
                } while( !(b->status & T_L) );

                wa->tcount++;
                t1 = t;
                t = t->layer[0].nextf;
                t1->hdr.status = 0;
        }

out:
        wa->conf_q = t;
        if( !wa->conf_q )
                wa->req_q = 0;

        return(t1);
}


/*********************************************************
 * routine: transmit_error
 *
 * description:
 *  Handle transmit error on an SCC channel.
 *
 * arguments:
 *  wa     points to work area
 *  id     the channel involved
 *
 * return code:
 *
 * side effects:
 *
 *********************************************************/
transmit_error(wa)
register WORK_AREA *wa;
{
        register T_FRAME *t, *t1;
        register IMP_BD *bd;
        register IMP_BD *retx;  /* first BD for retransmission */
        register int first;

        wa->terr_cnt++;

        /*
         * the IMP may have set the error bits in the
         * middle of a frame.
         * if so, clear all Ready bits until the end
```

```
        * of the frame and re-write the IMP's tx pointer
        */
        retx = bd = wa->imp_cbd;
        first = 0;

        if( bd->status & T_R ){
                issue_cmd(wa->restart);
                return;
        }

        /* find bd for which error was reported */
        while( !(bd->status & T_ERROR) ){
                first = bd->status & T_L;
                if( bd->status & T_W )
                        bd = wa->pram->tbd;
                else
                        bd++;
                if( first ) retx = bd;
                /*
                 * no error bits are set:
                 * we must have already taken the faulty buffer
                 */
                if( bd == wa->imp_cbd )
                        return(0);
        }

        /* now clear Ready bits until end of frame */
        bd = retx;
        while( !(bd->status & T_R) ){
                bd->status &= ~T_ERROR;
                if( bd != retx )
                        bd->status |= T_R;
                if( bd->status & T_L )
                        break;
                if( bd->status & T_W )
                        bd = wa->pram->tbd;
                else
                        bd++;
        }
        retx->status |= T_R;

        /* now write IMP's internal tx pointer */
        wa->pram->pscc.h.tbuf = (int)retx - (int)wa->pram;

        issue_cmd(wa->restart);
}
```

```
/*********************************************************
 * File:    scp.c
 *
 * Description:
 *  The driver routines for handling an SCP port.
 *
 * Routines:
 *  scp_init
 *  scp_txreq
 *  scp_rxbuf
 *
 * Author:
 *  Enghe Chimood
 *********************************************************/

#include "system.h"
#include "config.h"
#include "register.h"
#include "imp.h"


extern P_SCP_SMI_PRAM p_Spram;
extern P_SCP_SMI_MODE p_Smode;


/*********************************************************
 * routine: scp_txreq
 *
 * description:
 *  Transmit a byte of data on the SCP channel.
 *
 * arguments:
 *  data       the data byte for transmission
 *
 * return code:
 *  -1      channel not ready
 *  0       on success
 *
 * side effects:
 *
 *********************************************************/
scp_tx_rx (tx_data, p_rx_data)
BYTE tx_data;
p_BYTE p_rx_data;
{
WORD i, j;
        /* Done bit was not cleared */
        if( p_Spram->scp_bd & SCP_R ){
                return(-1);
        }

        p_Spram->scp_bd = *tx_data;
        p_Spram->scp_bd |= SCP_R;

        SCP_ENABLE;
        /* transmission requested via mode register */
        p_Smode->scp_start = 1;
```

```
        j = 0;
        /* now, check the data rx result */
        while ( p_Spram->scp_bd & SCP_R )
        {
                for (i = 0; i < 0x7f; i++)
                ;

                if (j++ >= 5)
                        return (-1);
        }

        *p_rx_data = p_Spram->scp_bd & 0xff;
        SCP_DISABLE;

        return(S_OK);
}


scp_nb_read(tx_data, p_rx_data)
BYTE tx_data;
p_BYTE p_rx_data;
{
        tx_data |= U_READ;
        scp_tx_rx (tx_data, p_rx_data);
        *p_rx_data &= 0x0f;
}

scp_nb_write (tx_data)
BYTE tx_data
{
BYTE rx_data;

        tx_data |= U_WRITE;
        scp_tx_rx (tx_data, &rx_data);
}

scp_nr6_read (addr, p_rx_data)
BYTE addr;
WORD *p_rx_data;
{
BYTE tmp_data;

        addr |= U_READ;
        scp_tx_rx (addr, &tmp_data);

        *p_rx_data |= (tmp_data & 0x0f);
        *p_rx_data = *p_rx_data << 8;                   /* shift to high byte */

        /* need to read a second time */
        scp_tx_rx (addr, &tmp_data);
        *p_rx_data |= tmp_data;
}

scp_nr6_write (addr, tx_data)
BYTE addr;
BYTE tx_data;
{
```

```
BYTE dummy;

        addr |= U_WRITE;
        scp_tx_rx (addr, &dummy);
        scp_tx_rx (tx_data, &dummy);
}

scp_br_read (addr, p_rx_data)
BYTE addr;
p_BYTE p_rx_data;
{
        addr |= U_READ;
        scp_tx_rx (addr, p_rx_data);

        /* first byte read should be garbage */
        /* need to read a second time */
        scp_tx_rx (addr, p_rx_data);
}

scp_br_write (addr, tx_data)
BYTE addr;
BYTE tx_data;
{
BYTE dummy;

        addr |= U_WRITE;
        scp_tx_rx (addr, &dummy);
        scp_tx_rx (tx_data, &dummy);
}

Nt_init ()
{
        scp_br_write (BR0, 0x7f);
        scp_br_write (BR2, 0xf0);
        scp_br_write (BR9, 0xcc);

        scp_nb_write (NR4 | 0x0f);  /* enable all interrupt */
}

Lt_init ()
{
        scp_br_write (BR0, 0x7f);
        scp_br_write (BR2, 0xf0);
        scp_br_write (BR9, 0xcc);

/* EC debug
 * the R6 eoc value is not set.
 */

        scp_nb_write (NR4 | 0x0f);  /* enable all interrupt */
}

scp_isr ()
{
T_NR3 nr3_data;

        scp_nb_read (NR3, (p_BYTE) &nr3_data);
```

```
        if (lcb->local_mode == NT)
        {
                Nt_isr (nr3_data);
        }
        else
        {
                Lt_isr (nr3_data);
        }

        CLEAR_IPA;
}

Nt_isr (irq_data)
T_NR3 irq_data;
{
        if (irq_data.nr1_chg)
        {
        }

        if (irq_data.eoc_chg)
        {

        }

        if (irq_data.m4_chg)
        {
        }

        if (irq_data.m56_chg)
        {

        }
}

Lt_isr (irq_data)
T_NR3 irq_data;
{

@
```

```
/*********************************************************
        General type define and constants
*********************************************************/

/* Positions of bits within a byte */
#define BIT0 0x1
#define BIT1 0x2
#define BIT2 0x4
#define BIT3 0x8
#define BIT4 0x10
#define BIT5 0x20
#define BIT6 0x40
#define BIT7 0x80

/* TYPE DEFINITION */

typedef enum T_BOOLEAN {FALSE, TRUE} BOOLEAN;

/* some system wide data type definitions */

typedef unsigned char  BYTE, *p_BYTE; /*  8 bit unsigned      */
typedef char           CHAR, *p_CHAR; /*  8 bit signed        */
typedef unsigned short WORD;      /* 16 bit unsigned      */
typedef short int      INTEGER;   /* 16 bit signed        */
typedef unsigned long  LONGWORD;  /* 32 bit unsigned      */
typedef long int       LONGINT;   /* 32 bit signed        */
typedef unsigned       BIT;       /* bitfield (16/32)     */
typedef unsigned short STATUS_CODE; /* type for status return */


/* ISTA Control Block */
typedef struct
{
        BYTE   cur_state; /* current ISTA state */
#define RESET_ST      0  /* system in reset */
#define READY_ST      1  /* system ready for activation */
#define LOCAL_ACT     2  /* Activation detected locally,
                                                        pending remote activation */
#define REMOTE_ACT    3  /* remote activated, pending local activation */
#define DATA_TRANSFER 4  /* channel established */
#define LOCAL_DEACT   5  /* Local deactivation detected */
#define REMOTE_DEACT  6  /* remote deactivated */

        BYTE   local_mode; /* NT or LT mode */
#define IS_NT  0x0
#define IS_LT  0x01

} ICB;

/*
 * Type definition for SCC0, SCC1 interrupt event table entry
 */
typedef struct
{
        union {
                struct {
                        WORD   evt;
                        WORD   port;
```

```
                } scc;
                struct {
                        WORD   u_reg;
                        BYTE   evt;
                        BYTE   res;
                } ui;
                LONGWORD lw;
        } u;
} INTR_EVENT;

#define MAX_EVT_ENTRY   16
```