GROUND TERMINAL SIMULATOR IMPLEMENTATION FOR UPLINK SYNCHRONIZATION TRIALS

by

Caroline Tom

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
REPORT NO. 1341

November 1998
Ottawa

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited
GROUND TERMINAL SIMULATOR IMPLEMENTATION
FOR UPLINK SYNCHRONIZATION TRIALS

by

Caroline Tom
Military Satellite Communications Group
Space Systems and Technology Section

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
REPORT NO. 1341

PROJECT
5CA11

November 1998
Ottawa
Abstract

A ground terminal (GT) simulator was developed at Defence Research Establishment Ottawa (DREO) as part of an in-house activity examining aspects of uplink synchronization for extremely high frequency (EHF) satellite communications (SATCOM) using frequency hopping. The GT simulator consists of a GT processor, custom interface boards, synthesizer controller, frequency synthesizer, and data source. The GT processor is the principal component of the simulator and is realized by a TMS320C30 digital signal processor board. This report describes the implementation of the GT processor functions relating to uplink synchronization and the interfaces between the various components of the simulator. This report also describes the synchronization procedure for the GT simulator. The procedure is broken down into three steps: downlink synchronization; uplink coarse synchronization; and uplink fine synchronization. A guide on the hardware installation of the various components of the GT simulator and a list of the software needed to run the simulator is provided in an appendix.

Résumé

Un simulateur de terminal au sol a été développé au Centre de Recherches pour la Défense à Ottawa (CRDO). Ce simulateur fait partie d’un travail au CRDO concernant les aspects de synchronisation de la liaison sol-espace pour les communications par satellite, utilisant le spectre étalé dans la bande de fréquence extrêmement haute. Le simulateur consiste d’un processeur de terminal au sol, des cartes d’interfaces fabriquées sur demande, un contrôleur de synthétiseur, un synthétiseur de fréquences, et une source de données. Le processeur du terminal au sol est la pièce principale du simulateur. Le rôle de processeur du terminal au sol est réalisé par une unité de traitement de signaux numériques, TMS320C30. Ce rapport décrit la réalisation des fonctions du processeur de terminal au sol liées à la synchronisation de la liaison sol-espace. Ce rapport décrit aussi la procédure pour la synchronisation du simulateur de terminal au sol. La procédure est divisée en 3 étapes: la synchronisation de la liaison espace-sol; la synchronisation préliminaire de la liaison sol-espace; et la synchronisation précise de la liaison sol-espace. Un guide d’installation des pièces du simulateur de terminal au sol est fourni ainsi qu’une liste des logiciels requis pour le fonctionnement du simulateur.
Executive Summary

The Military Satellite Communications (MSC) Groups at Defence Research Establishment Ottawa (DREO) and Communications Research Centre (CRC) have been examining synchronization aspects of robust, anti-jam satellite communications at extremely high frequency (EHF). The MSC groups at DREO and CRC cooperatively developed ground terminal (GT) and payload simulators to carry out synchronization trials over the United Kingdom (UK) Skynet 4A EHF transponder. The use of the Skynet 4A EHF transponder was made possible through a Memorandum of Understanding established under The Technical Cooperation Program (TTCP). With the Skynet 4A EHF transponder, the ground-based GT and payload simulators are set up to realize a single path (either uplink or downlink) of a practical EHF communications link.

Before communications of a frequency hopped system can begin, synchronization of the GT and payload must be performed. The first step in the synchronization procedure is downlink synchronization, followed by uplink coarse synchronization, and finally uplink fine synchronization. The in-house activity examined downlink and uplink synchronization separately. The first part of the in-house activity focussed on downlink synchronization aspects and has been documented. During downlink synchronization, the GT gathers information about the system clock to allow it to proceed to uplink synchronization. In uplink synchronization, the GT attempts to align its clock with the payload by transmitting synchronization probes at designated times. The payload receives the probes and formulates synchronization responses. The responses are transmitted back to the GT and are used to adjust the GT clock.

This report describes the development of a GT simulator for the uplink synchronization trials. The GT simulator consists of a GT processor, a number of custom interface boards, frequency synthesizer, RF equipment, and a data source. The GT processor was implemented on a TMS320C30 digital signal processor (DSP) board and is contained in a host PC. The custom interface boards include a GT processor interface board which generates the necessary clock signals for the GT processor and provides the interface between the GT processor and the hopping synthesizer controller (HSC). The HSC, in turn, controls a frequency synthesizer. A multipurpose data interface board was also designed and fabricated to provide the interface between the GT processor and a data source, and to provide the interface to a downlink synchronization reference link.

The modes of operation for the GT simulator include: transmitting a continuous-wave (CW) tone at specific points in the hopping bandwidth; transmitting an arbitrary CW tone within the hopping bandwidth; sweeping a CW tone across the hopping bandwidth; performing downlink synchronization; performing uplink coarse synchronization; and performing uplink fine synchronization. The first three functions were implemented during the development of the simulator and are retained for debug purposes. The GT simulator has been developed so that commands for the various modes of operation can be issued remotely once the GT simulator is powered on and the executable file is run. In this implementation, remote operation of the GT
simulator is carried out by the payload simulator. The remote operation capability is included to facilitate the system integration and trials since the GT and payload simulators are physically located 1.5 km apart.

Although the focus of the second part of the in-house activity is only on uplink synchronization, there is a need to provide a mechanism for establishing downlink synchronization beforehand. As the experimental setup for the uplink synchronization only realizes a single path of an EHF communications link, a simulated downlink was set up between the GT and payload simulators using an RS232 serial communications link. During downlink synchronization, a downlink synchronization reference pulse is continuously transmitted by the payload to the GT simulator. The edges of the reference pulse correspond to specific instances in the pseudorandom hopping sequence of the payload simulator. The GT simulator detects and uses the pulse to form a preliminary estimate of the system clock.

Once downlink synchronization is achieved, the GT simulator can proceed with uplink coarse synchronization. In coarse synchronization, two consecutive bursts of sixteen synchronization probes are transmitted by the GT simulator using different timing hypotheses. In this implementation, an "outward moving" search scheme is used to test different timing hypotheses. The search scheme consists of starting at the most probable hypothesis obtained during downlink synchronization and shifting each subsequent timing hypothesis outward on either side of the most probable hypothesis. The payload receives the coarse synchronization probes and formulates a binary "detect/no detect" response. The response is relayed back to the GT simulator via another serial link. When a "detect" is received by the GT simulator, a verification process is carried out to ensure it is a valid "detect" response. Coarse synchronization is considered to be achieved when the GT clock is aligned to within a hop of the payload clock.

In fine synchronization, a single burst of thirty-two synchronization probes is transmitted at designated times. Again, the payload simulator receives the probes and computes a synchronization estimate to be returned to the GT simulator. The synchronization estimate represents how early or late the received probes are relative to the system clock. In order to reduce the number of times the GT clock is adjusted and to minimize any estimate errors due to noise, an average of ten fine synchronization responses is used to determine the final adjustment of the GT clock. The fine synchronization routine is repeated until the GT clock is aligned to within 10% of a hop of the payload clock.

The procedures for setting up the GT simulator are included in Appendix A. The listings of programs used by the GT simulator and the simulation parameter data files are contained in Appendix B and Appendix C respectively.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>RÉSUMÉ</td>
<td>iii</td>
</tr>
<tr>
<td>EXECUTIVE SUMMARY</td>
<td>v</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF SYMBOLS AND ABBREVIATIONS</td>
<td>xv</td>
</tr>
<tr>
<td>1.0 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Task Description</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Report Outline</td>
<td>2</td>
</tr>
<tr>
<td>2.0 Synchronization Procedure</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Downlink Synchronization</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Uplink Synchronization</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Uplink Coarse Synchronization</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Uplink Fine Synchronization</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Synchronization Responses</td>
<td>9</td>
</tr>
<tr>
<td>3.0 System Description</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Simulator Setup</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Ground Terminal Simulator Subsystem Hardware</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Ground Terminal Processor</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Ground Terminal Processor Interface Board</td>
<td>13</td>
</tr>
<tr>
<td>3.2.3 Hopping Synthesizer Controller and Frequency Synthesizer</td>
<td>13</td>
</tr>
<tr>
<td>3.2.4 Data Device and Multipurpose Data Interface Board</td>
<td>13</td>
</tr>
<tr>
<td>3.2.5 Synchronization Response Return Serial Link</td>
<td>13</td>
</tr>
<tr>
<td>3.2.6 Downlink Synchronization Reference Serial Link</td>
<td>14</td>
</tr>
<tr>
<td>3.2.7 Hardware Interface Requirements of the GT Simulator</td>
<td>14</td>
</tr>
<tr>
<td>3.2.7.1 DSPLINK Backplane Interface</td>
<td>14</td>
</tr>
<tr>
<td>3.2.7.2 Serial Link Interfaces</td>
<td>15</td>
</tr>
<tr>
<td>3.2.7.3 Data Source Interface</td>
<td>16</td>
</tr>
<tr>
<td>3.2.7.4 HSC Command and Transmit Data Interface</td>
<td>17</td>
</tr>
<tr>
<td>3.2.7.5 Frequency Synthesizer Interface for the HSC</td>
<td>17</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Ground Terminal Simulator Software</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 DSP Assembly Language Programs</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.1 Main Assembly Language Program</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.1.1 Preliminary Initialization by the GT Processor DSP</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.1.2 GT Simulator Modes of Operation</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1.2 Coarse Synchronization Assembly Routine</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1.2.1 Starting the Coarse Synchronization Procedure</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1.2.2 Coarse Synchronization Probe Generation</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1.2.3 Verification of &quot;Detect&quot; Responses Received</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1.2.4 Preparing for Fine Synchronization</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1.2.5 Clearing the Synchronization Response Buffer</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1.2.6 Search Range for Coarse Synchronization Routine Exceeded</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1.2.7 Synchronization Response Buffer Overflow</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1.3 Fine Synchronization Assembly Routine</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1.3.1 Beginning Fine Synchronization</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1.3.2 Generation of Fine Synchronization Probes</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1.3.3 Refining the Alignment of the Ground Terminal Clock</td>
<td>30</td>
</tr>
<tr>
<td>3.3.1.3.4 Nonconvergence of Fine Synchronization Estimates</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.3.5 Achieving Fine Synchronization</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.4 Interrupt Service Routine</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.4.1 ISR Housekeeping</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.4.2 ISR Tasks for Coarse Synchronization Mode</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1.4.3 ISR Tasks for Fine Synchronization Mode</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1.4.4 ISR Tasks for Downlink Synchronization Mode</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 Host/User Interface Program</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2.1 DSP Board Initialization</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2.2 Downloading Simulation Parameters</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2.3 Serial Communications Initialization</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2.4 User Interface Menu</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2.5 Host/User Interface Loop</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2.5.1 Checking for User Input - Local Keyboard Input</td>
<td>37</td>
</tr>
<tr>
<td>3.3.2.5.2 Checking for User Input - Remote Input</td>
<td>37</td>
</tr>
<tr>
<td>3.3.2.5.3 Checking for FR0 Pulse Edge Detection</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2.5.4 Checking for Completion of Task by DSP</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2.5.5 Checking for Return Link Synchronization Response</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2.5.6 Checking Other Flag Conditions</td>
<td>38</td>
</tr>
<tr>
<td>3.3.3 ASCII Data Files</td>
<td>39</td>
</tr>
<tr>
<td>4.0 Summary</td>
<td>41</td>
</tr>
<tr>
<td>References</td>
<td>43</td>
</tr>
</tbody>
</table>
# Table of Contents

## Appendix A - Ground Terminal Simulator Installation Guide
- A1 Installation ................................................................. A1
- A1.1 Hardware Installation .................................................. A1
- A1.2 Program Files .............................................................. A2

## Appendix B - Software Listings
- B1 GT Simulator Host/User Interface Program ....................... B1
- B2 DSP Main Program ......................................................... B17
- B3 Coarse Synchronization Routine ...................................... B33
- B4 Fine Synchronization Routine ......................................... B50
- B5 DSP Interrupt Service Routine ........................................ B57
- B6 TMS Linker .cmd File ..................................................... B62

## Appendix C - ASCII Data Files
- C1 General ........................................................................ C1
- C2 Freq.dat File ................................................................. C1
- C3 Hscinit.dat File ............................................................... C2
- C4 Gtparam.dat File ............................................................ C3
List of Figures

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Downlink transmission structure</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Downlink synchronization reference pulse detail</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>General time-frequency plan for uplink transmission structure</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Data flow of the coarse synchronization process</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Data flow of the fine synchronization process</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Synchronization response format</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>System block diagram of the uplink synchronization experiments</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>DSPLINK backplane interface connector</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Pinout configuration for serial connector to downlink synchronization</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>reference pulse</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Coarse synchronization procedure state diagram</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Search scheme for coarse synchronization timing hypotheses</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Different scenarios for GT clock alignment with payload clock</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Fine synchronization procedure state diagram</td>
<td>29</td>
</tr>
<tr>
<td>3.8</td>
<td>Interrupt service routine flow diagram</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>Host/user interface flow diagram</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>User menu for the GT simulator</td>
<td>35</td>
</tr>
<tr>
<td>3.11</td>
<td>Host/user interface loop operation</td>
<td>37</td>
</tr>
<tr>
<td>A.1</td>
<td>GT simulator TMS320C30 linker process</td>
<td>A2</td>
</tr>
<tr>
<td>A.2</td>
<td>GT simulator host PC linker process</td>
<td>A3</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>DSPLINK backplane interface pinout description</td>
<td>16</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>GT simulator modes of operation</td>
<td>19</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Description of flags for GT simulator</td>
<td>40</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ADJ_NCO</td>
<td>Adjust NCO state in fine synchronization routine</td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
<td></td>
</tr>
<tr>
<td>CLR_RESP_PIPE</td>
<td>Clear Response Buffer Pipeline state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>CRC</td>
<td>Communications Research Centre</td>
<td></td>
</tr>
<tr>
<td>CSYNC</td>
<td>Coarse Synchronization mode command of the HSC</td>
<td></td>
</tr>
<tr>
<td>CW</td>
<td>Continuous Wave</td>
<td></td>
</tr>
<tr>
<td>DIB</td>
<td>Data Interface Board</td>
<td></td>
</tr>
<tr>
<td>DP</td>
<td>Data Page</td>
<td></td>
</tr>
<tr>
<td>DREO</td>
<td>Defence Research Establishment Ottawa</td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
<td></td>
</tr>
<tr>
<td>EHF</td>
<td>Extremely High Frequency</td>
<td></td>
</tr>
<tr>
<td>FCALC</td>
<td>Frequency Calculate command for the HSC</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
<td></td>
</tr>
<tr>
<td>FINE_NT_ACH</td>
<td>Fine Synchronization Not Achieved state in fine synchronization routine</td>
<td></td>
</tr>
<tr>
<td>FR0</td>
<td>Frame 0</td>
<td></td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
<td></td>
</tr>
<tr>
<td>FSK/FRAME</td>
<td>Transmit data port of the GT processor i/f board</td>
<td></td>
</tr>
<tr>
<td>GEN_PROBES</td>
<td>Generate Probes state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>GO_2_FSYNC</td>
<td>Go to Fine Synchronization state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>GO_2_RUN</td>
<td>Go to RUN state in fine synchronization routine</td>
<td></td>
</tr>
<tr>
<td>GT</td>
<td>Ground Terminal</td>
<td></td>
</tr>
<tr>
<td>HSC</td>
<td>Hopping Synthesizer Controller</td>
<td></td>
</tr>
<tr>
<td>i/f</td>
<td>Interface</td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td></td>
</tr>
<tr>
<td>INIT_SECTION</td>
<td>Initialization Section in fine synchronization routine</td>
<td></td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
<td></td>
</tr>
<tr>
<td>JDLS</td>
<td>Joint Data Link Standard</td>
<td></td>
</tr>
<tr>
<td>MOU</td>
<td>Memorandum Of Understanding</td>
<td></td>
</tr>
<tr>
<td>MSC</td>
<td>Military Satellite Communications</td>
<td></td>
</tr>
<tr>
<td>NCO</td>
<td>Numerically-Controlled Oscillator</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
<td></td>
</tr>
<tr>
<td>PL</td>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>PLINE_ERR</td>
<td>Pipeline Overflow Error state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>PRELIM_INIT</td>
<td>Preliminary Initialization state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>proc</td>
<td>Processing</td>
<td></td>
</tr>
<tr>
<td>prop</td>
<td>Propagation</td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
<td></td>
</tr>
<tr>
<td>SATCOM</td>
<td>Satellite Communications</td>
<td></td>
</tr>
<tr>
<td>SHF</td>
<td>Super High Frequency</td>
<td></td>
</tr>
<tr>
<td>SRCH_EXCEED</td>
<td>Search Range Exceeded state in coarse synchronization routine</td>
<td></td>
</tr>
<tr>
<td>TTCP</td>
<td>The Technical Cooperation Program</td>
<td></td>
</tr>
<tr>
<td>TXOFF</td>
<td>Transmit off command bit for the HSC</td>
<td></td>
</tr>
<tr>
<td>TX_FPROBES</td>
<td>Transmit Fine Synchronization Probes in fine synchronization routine</td>
<td></td>
</tr>
</tbody>
</table>
### List of Symbols and Abbreviations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>ULGO</td>
<td>Uplink GO command for the HSC</td>
</tr>
<tr>
<td>VER_DETECT</td>
<td>Verify Detect state in coarse synchronization routine</td>
</tr>
</tbody>
</table>
1.0 Introduction

1.1 Background

The Military Satellite Communications (MSC) Group at Defence Research Establishment Ottawa (DREO), along with its sister group at Communications Research Centre (CRC) have been examining aspects of robust, anti-jam satellite communications at extremely high frequency (EHF). An area critical to the operation of EHF frequency hopped satellite communications (SATCOM) systems is the synchronization of the ground terminal (GT) clock to the payload system clock. In order to gain a better appreciation of the processes involved in synchronization, the MSC groups at DREO and CRC developed GT and payload simulators to carry out synchronization trials. Trials were carried out over the United Kingdom (UK) Skynet 4A EHF transponder, made possible through a memorandum of understanding (MOU) established under The Technical Cooperation Program (TTCP). The Skynet 4A transponder receives an EHF signal and translates it to a super high frequency (SHF) signal which is subsequently retransmitted. Consequently, the ground-based simulators for the GT and the payload are used to realize a single path of a practical EHF communications link.

Before communications of an EHF system can begin, synchronization of the GT and payload must be performed. Downlink synchronization is considered to be the first step in the synchronization process and was the focus of the first part of the in-house activity examining EHF satellite communications. The work on downlink synchronization was documented in [1]. During downlink synchronization, the GT gathers synchronization information about the system clock to allow it to receive frequency hopped signals from the satellite. The synchronization information also provides a basis to begin uplink synchronization. The GT simulator performs uplink synchronization in order to align its clock with the payload clock after taking into account the non-deterministic propagation delay. Processing delays, although small by comparison, can also be factored into the alignment of the GT clock. Once uplink synchronization is achieved, the GT simulator can begin to transmit user data over the frequency hopped SATCOM system.

The uplink synchronization process consists of two phases: coarse synchronization and fine synchronization. In both phases, the GT simulator transmits synchronization probes to the payload. The payload receives the probes and formulates a synchronization response for the GT simulator. The synchronization response during coarse synchronization indicates whether the probes are detected by the payload simulator. In fine synchronization, the synchronization response reflects how early or late the received probes are, relative to the payload clock. The synchronization responses are relayed to the GT simulator and are used by the GT simulator to adjust its clock. The aspects of uplink synchronization were the focus of the second part of the in-house activity and are described in this report.

Users of an EHF satcom system communicate according to a data link standard which defines the uplink and downlink signal formats and the processing operations. For the in-house activity, a Joint Data Link Standard (JDLS) was written [2] to provide the parameters for uplink
and downlink transmission structures. The parameters include the modulation scheme, data rate, frame and channel sizes, and subframe allocations.

1.2 Task Description

In order to carry out uplink synchronization experiments, GT and payload simulators had to be developed. Each of the simulators consists of a processing unit which performs the synchronization tasks and interfaces to other components of the simulator. The payload processor must receive, demodulate, and process transmitted synchronization probes. The payload processor must also formulate responses corresponding to the synchronization probes received and transmit the responses back to the GT simulator. In addition, the payload processor must generate a synchronization aid which is used by the GT simulator to establish a starting point for performing uplink synchronization. The generation of a synchronization aid is intended as a substitute for the downlink synchronization process described above since only the uplink path can be realized using the transponding Skynet satellite. For the purposes of the in-house activity and of this report, the detection of the synchronization aid by the GT simulator is referred to as performing downlink synchronization. The payload processor also interacts with other hardware components of the simulator. The hardware components include a hopping synthesizer controller (HSC) which controls a frequency synthesizer, a data interface board which connects to a data sink, and a first-in-first-out (FIFO) interface board which holds the samples of the received signal.

The GT processor performs reciprocal tasks of the payload processor. The GT processor assembles and transmits modulated uplink synchronization probes at specific allocated times according to the data link standard. The GT processor must also receive and decode the synchronization responses during uplink synchronization. As well, the GT processor is responsible for detecting the synchronization aid generated by the payload simulator. Furthermore, the GT processor interacts with other components of the GT simulator. Once such component is a GT processor interface (IF) board which generates the necessary GT clock signals and transfers commands to an HSC. The HSC, in turn, controls a transmittting frequency synthesizer. The GT processor also communicates with a data interface board which is connected to the data source.

1.3 Report Outline

The purpose of this report is to describe the development of the GT simulator for the uplink synchronization experiments. A detailed description of the synchronization procedure is provided in Section 2.0. In Section 3.0, the simulator setup is described to introduce the components of the GT simulator. Subsequently, the description of the GT simulator is broken down into the hardware and software features of the simulator. The hardware section deals mainly with the physical components and interfaces. The software section describes the tasks performed by the GT simulator. Three appendices are included in this report. Appendix A consists of a guide which outlines the installation and setup procedures of the GT simulator.
Appendix B contains a listing of all the programs used for the GT simulator. Appendix C contains a copy of the American Standard Code for Information Interchange (ASCII) data files used by the GT simulator to download specific simulation parameters.
2.0 Synchronization Procedure

2.1 Downlink Synchronization

The first step in establishing communications over an EHF frequency hopped SATCOM system consists of achieving downlink synchronization. The purpose of performing downlink synchronization is to acquire the satellite downlink including demodulating user data and synchronization information. The downlink synchronization process precedes uplink synchronization. From the data link standard described in [2], the payload terminal transmits synchronization hops based on a time division multiplex scheme. The GT simulator, upon receiving and detecting the synchronization hops, is then able to derive an estimate of the payload (system) clock in order to begin uplink synchronization. The general downlink transmit structure is shown in Fig. 2.1. A 20 ms frame consisting of 320 hops is further divided into 40 time slots of 8 hops each. The data link standard specifies that the first 36 time slots are allocated to user data. The next two are reserved and the last two are used for synchronization purposes.

![Downlink transmission structure](image)

**Fig. 2.1 Downlink transmission structure**

As described in Section 1.1, the uplink synchronization trials are carried over the Skynet 4A EHF transponder. As a transponding satellite is used, only one path (uplink or downlink) of an EHF satcom system with onboard processing can be realized at any one time. The use of a transponding satellite for the uplink synchronization trials also means both the GT simulator and payload simulator are developed as ground-based systems. For the uplink synchronization trials, a downlink “path” is simulated by a direct serial link connection between the payload and GT simulators. Furthermore, to facilitate the implementation of the simulated downlink, a reference pulse is transmitted in lieu of synchronization hops to transmit a reference of the payload terminal clock (master clock). The reference pulse is referred to as the downlink synchronization reference pulse and is shown in Fig. 2.2. The edges of the reference pulse were chosen to correspond to specific points in the pseudorandom hop sequence. The rising edge of the downlink synchronization reference pulse corresponds to the start of hop number 0 of frame number 0 in the pseudorandom sequence. The falling edge of the reference pulse corresponds to the start of hop number 0 of frame number 1. The GT simulator receives this reference pulse and resets its own hop clock accordingly in preparation for uplink synchronization. For the uplink
synchronization experiments, the downlink synchronization reference pulse is continuously generated and transmitted by the payload simulator.

![Diagram of downlink synchronization reference pulse detail](image)

**Fig. 2.2 Downlink synchronization reference pulse detail**

### 2.2 Uplink Synchronization

Once downlink synchronization is achieved, the GT simulator can proceed with uplink synchronization. Uplink synchronization is the process whereby the GT simulator attempts to align its own clock with the payload (system) clock. The uplink synchronization process is carried out independently from downlink synchronization because in uplink synchronization, the ground terminal takes into account the propagation delay so that its transmissions are received at the appropriate time by the payload. The GT simulator transmits bursts of synchronization probes which correspond to different timing offset hypotheses. The payload simulator receives the probes and produces a response. The response indicates the coarse synchronization probes detection and the fine synchronization probe timing offset relative to the payload clock in a particular frame. The GT simulator then uses the responses to align its clock with the payload clock.

The timing offset hypotheses of the transmitted synchronization probes correspond to specific instances relative to the time-frequency plan described in [2] and illustrated in Fig. 2.3. The format of the time-frequency plan is based on a multichannel subframe/frame structure which is repeated on a frame basis. A subframe for a particular channel is referred to as a cell [2]. There are four channels implemented for the uplink synchronization trials. Each channel is subdivided into 8 frequency tone bins to support the 8-ary frequency-shift-keying (FSK) uplink modulation scheme specified in [2]. Each cell is designated as either a data cell or a synchronization cell for transmission of data or synchronization probes respectively. A particular user would be assigned specific cells to transmit data or synchronization probes.
The uplink synchronization process is carried out in two stages. The first stage is referred to as coarse synchronization. Coarse synchronization is achieved when the payload detects the GT simulator's synchronization probes. For the uplink synchronization trials, coarse synchronization occurs when the GT simulator clock is within a hop of the payload clock. The second stage of uplink synchronization is called fine synchronization. During fine synchronization, the GT simulator attempts to refine the alignment of its clock to come within 10% of a hop. The method for carrying out coarse and fine synchronization are described further in the following subsections.

2.2.1 Uplink Coarse Synchronization

Coarse synchronization probes are transmitted in a specific cell in the time-frequency plan and are composed of two contiguous bursts of sixteen probes [2]. The synchronization probes are located in a specific FSK tone bin. The probes are subsequently frequency hopped for transmission. In this implementation, the coarse synchronization probes are generated by a frequency synthesizer which is controlled by the HSC [3]. Each sixteen probe burst has a different timing hypothesis. The procedure for generating the coarse synchronization probes is given in Section 3.3.1.2.2. The GT simulator transmits the coarse synchronization probes by issuing a command to the HSC to precompute the frequency hopped probe frequencies for the two bursts and by commanding the HSC to switch to those frequencies at an appropriate time. After a propagation delay and a processing delay, a binary response is returned by the payload simulator as a synchronization response indicating if the coarse synchronization probes were detected. Details of the synchronization response are provided in Section 2.2.4. The consistent detection of coarse synchronization probes with the same hypothesis indicates that the GT clock
is within a hop of the payload clock. Coarse synchronization is thus achieved. The general data flow for coarse synchronization is shown in Fig. 2.4.

![Diagram of Coarse Synchronization Probes]

**Fig. 2.4 Data flow of the coarse synchronization process**

2.2.2 Uplink Fine Synchronization

Fine synchronization probes are also transmitted in a specific FSK tone bin of a specific cell in the time-frequency plan. However, fine synchronization probes consist of a single burst of 32 probes rather than two bursts of 16 as in the coarse synchronization process. Again the synchronization probes are generated by a frequency synthesizer. Information on the specific cell (i.e. channel number and FSK tone bin) to be used for the fine synchronization probes is transmitted by the GT processor to the HSC at the appropriate time (subframe). The payload simulator processes the received probes and formulates a response which indicates how early or late the probes are in relation to the payload clock. Details of the synchronization response format are given in Section 2.2.4. For the uplink synchronization experiments, uplink synchronization is considered achieved when the GT clock is aligned to within 10% of the payload clock. The value of 10% was considered reasonable to account for frequency drift while causing minimal degradation to FSK modulation. A general data flow diagram for the fine synchronization process is shown in Fig. 2.5.
2.2.3 Synchronization Responses

Synchronization responses are formulated by the payload simulator each frame to provide feedback to the GT simulator on the detection of coarse synchronization probes and on the estimation of the time offset of fine synchronization probes. For the uplink synchronization experiments, a synchronization response is returned via a synchronization response return link. The return link is described in Section 3.2.5. The synchronization response return link is implemented using an RS232 serial connection between the GT and payload. The serial connection is capable of supporting communications at 9.6 kb/s. With one response being transmitted per 20ms frame, the serial communications link is able to support synchronization responses which are about 20 characters in length.

For the uplink synchronization trials, the synchronization response consists of a reference frame number, the coarse synchronization detection results for the two coarse synchronization probe channels, and the fine synchronization estimates for the two fine synchronization probe channels. The frame number ranges in value between 0 and 191. The coarse synchronization response for each synchronization burst is a binary response, i.e. it is either a “detect” or “no detect”. The “detect” and “no detect” responses are represented by “1” and “0” respectively. As there are two probe bursts per channel per frame, there are four possible combinations of detection responses for each channel. In this implementation, the detection result for the first probe burst is selected to occupy the most significant bit. The combinations of detection results are mapped to a decimal representation for the return link synchronization response and thus, are represented by a value between 0 and 3.

The fine synchronization response is a number between -31 and 31 representing the timing error of the GT clock. A negative fine synchronization response indicates that the GT clock is early while a positive fine synchronization estimate indicates that the GT clock is late. It is assumed that coarse synchronization is achieved prior to performing fine synchronization so that the GT clock is within a hop of the payload clock at the start of fine synchronization. As a
result, with a hop period of 62.5 µs, the fine synchronization estimate (timing error) represents approximately half the actual timing error, in µs.

In order to comply with the message length restrictions for the serial connection, it was decided that the synchronization response would be formatted as shown in Fig. 2.6. An example of a synchronization response received by the GT simulator is also included.

![Synchronization response format](image)

**Fig. 2.6 Synchronization response format**

For the example in Fig. 2.6, the GT simulator would subsequently decode the response as:

- **096** ➩ Synchronization response for frame number 96
- **2** ➩ Coarse synchronization response for channel 0
  \[2_{10} = 10_{2}\text{ “detect” for burst 0} \]
  \[0 \text{ “no detect” for burst 1}\]
- **1** ➩ Coarse synchronization response for channel 1
  \[1_{10} = 01_{2}\text{ “no detect” for burst 0} \]
  \[0 \text{ “detect” for burst 1}\]
- **+12** ➩ Fine synchronization response for channel 0, payload estimates that received probes are 24 µs later than payload clock
- **-04** ➩ Fine synchronization response for channel 1, payload estimates that received probes are 8 µs earlier than payload clock
3.0 System Description

3.1 Simulator Setup

A system block diagram of the simulator setup for the uplink synchronization experiments is shown in Fig. 3.1. The GT and payload simulators are ground-based systems which are located approximately 1.5 km apart on the DREO/CRC site.

Data to be transmitted from the GT is modulated using 8-ary FSK as specified in [2]. The GT processor transfers the modulated data to the GT processor i/f board [4]. The data is read by the HSC which calculates the frequency of the next hop. The result of the calculations is a frequency value which includes the modulated data and the hop frequency. The HSC then passes this frequency value to the frequency synthesizer. The frequency synthesizer output is then converted to the radio frequency (RF) transmit signal at EHF. Upon receiving the EHF transmitted signal, Skynet 4 translates and retransmits the signal at X-band to the payload terminal. The received signal is downconverted and transferred to a frequency synthesizer which is controlled by another HSC. The frequency synthesizer generates the hopping pattern used to dehop the received signal. A Fast-Fourier Transform (FFT)-based processor is then used to produce samples of the received signal for each FFT channel [5]. The samples are stored on a FIFO interface board until the payload processor is ready to process the data.

The entire process described above encompasses only the uplink portion of an actual EHF satcom system with on-board processing. In an actual system, when the payload receives and processes the data during synchronization, a response is formulated and transmitted back to the GT on the downlink. For the uplink synchronization trials, the downlink portion is simulated using a direct serial link called the “synchronization response return link” which will be used to transmit the responses generated by the payload processor to the GT. The synchronization response return link is separate from the downlink synchronization reference link described in Section 2.1 which is used to simulate downlink synchronization. Furthermore, the interface of the synchronization response return link to each simulator is through the host personal computer (PC). By contrast, the interface of the downlink synchronization reference pulse serial link to each simulator is provided through a multipurpose data interface board designed and fabricated at DREO [6]. A data source and data sink are included in the simulator setup for data communications once fine synchronization is achieved.
3.2 Ground Terminal Simulator Hardware

3.2.1 Ground Terminal Processor

The GT processor for the uplink synchronization experiments consists of a Texas Instruments’ TMS320C30 digital signal processor (DSP) board. The TMS320C30 board is installed in a single 16-bit slot of an IBM-compatible host PC with a monitor and keyboard. Communications between the PC and the DSP board go through the PC’s input/output (I/O) space. The GT processor communicates with other components of the GT simulator using the serial port of the PC or by using the DSPLINK interface supported by the TMS320C30 DSP board. These interfaces are discussed further in the subsections below.
3.2.2 Ground Terminal Processor Interface Board

A GT processor interface board was designed and fabricated at DREO for the uplink synchronization experiments and is documented in [4]. The board was designed to perform three functions: to generate the necessary clock signals for the GT simulator operations; to provide a command interface for a hopping synthesizer controller [3]; and to provide an interface for transferring FSK/channel data to the hopping synthesizer controller. Communications between the GT processor and the GT processor i/f board is achieved through the DSPLINK interface of the TMS320C30 DSP board. The GT processor i/f board is installed in a DSPLINK backplane chassis that was also assembled at DREO. The DSPLINK interface and backplane are described further in Section 3.2.7.1. Details on the specific operation of the GT processor i/f board can be found in [4].

3.2.3 Hopping Synthesizer Controller and Frequency Synthesizer

A frequency synthesizer is used in the uplink synchronization experiments to produce the appropriate frequency-hopped FSK tone to be transmitted by the GT. In this implementation of the GT simulator, two synthesizers are supported: the Comstron FS2000 Frequency Synthesizer [7]; and the Sciteq VDS-2G-469 Frequency Synthesizer [8]. Both frequency synthesizers are driven by an HSC which was developed at DREO [3]. The HSC, in turn, is controlled by the GT processor via the GT processor i/f board which is described above. The HSC receives and actions commands from the GT processor relating to its initialization and mode of operation. The HSC performs all the calculations required for frequency computation (including a random number generator routine for the pseudorandom hop sequence), and frequency word format conversion. Finally, the HSC transfers the resulting frequency word to the frequency synthesizer at the appropriate point in time. A detailed description of the HSC is provided by [3].

3.2.4 Data Device and Multipurpose Data Interface Board

The data source for the GT simulator subsystem consists of an HP1645A Bit-error-rate test set [9]. The interface between the HP1645A and the GT processor is realized using a multipurpose data interface board (DIB) which was developed at DREO and is documented in [6]. The DIB receives a single-bit RS232 data stream from the HP1645A, converts the data to TTL levels, and formats the data stream into 12-bit words to be read by the GT processor. The DIB is installed in the DSPLINK backplane chassis and communicates with the GT processor via the DSPLINK interface. The DSPLINK interface and backplane are discussed in Section 3.2.7.1.

3.2.5 Synchronization Response Return Serial Link

As mentioned in Section 2.2.3, the payload simulator formulates a synchronization response for every frame. The synchronization response contains information on whether any
probes were detected in the coarse synchronization cells which are shown in Fig. 2.3. The synchronization response also contains estimates of the timing offset of probes which may be received in the fine synchronization cells. In a practical EHF satcom system with on-board processing, the synchronization response is transmitted by the payload to the GT on the downlink. For the uplink synchronization experiments, a transponding satellite is used and thus, another means of transmitting the synchronization responses to the GT simulator is required. A synchronization response return link is realized using an RS232 serial connection between the GT and payload ground-based simulators. The serial connection supports communications at 9.6 kb/s and is accessed through the serial connector of the PC host for each simulator. Serial communications software [10] developed at DREO is used by the payload and GT hosts to send and retrieve the synchronization responses respectively.

3.2.6 Downlink Synchronization Reference Serial Link

A downlink synchronization reference serial link is implemented to transmit the downlink synchronization reference pulse which is described in Section 2.1. The downlink synchronization reference pulse is analogous to using synchronization hops of a real processing satellite. Similar to the case for the relaying of synchronization responses back to the GT, another method of transmitting the reference pulse is required as a result of a transponding satellite being used for the trials. In order to be able to present the reference pulse to the GT processor in real time, it was decided to route the downlink synchronization reference serial link by way of the DIB. By using the DIB, it is possible to avoid having to use the slower host/DSP interface. Details of the downlink synchronization reference serial link implementation are found in [6].

3.2.7 Hardware Interface Requirements of the GT Simulator

3.2.7.1 DSPLINK Backplane Interface

In order to support multiple custom boards using the DSPLINK interface to communicate with the GT processor, a DSPLINK backplane chassis was assembled for the uplink synchronization experiments. Currently, the backplane chassis houses the GT processor i/f board, the DIB, and an adaptor board to map the 50-pin DSPLINK interface [11] connector from the DSP board to the 96-pin backplane connector. The DSPLINK backplane connector and pinout description are shown in Fig. 3.2 and Table 3.1 respectively.
Corresponding signal names

<table>
<thead>
<tr>
<th>PIN</th>
<th>ROW A</th>
<th>ROW B</th>
<th>ROW C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HOP CLK</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HOP CLK*</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RESERVED</td>
<td>D3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RESERVED</td>
<td>D4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RESERVED</td>
<td>D5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RESERVED</td>
<td>D6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>RESERVED</td>
<td>D7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>GND</td>
<td>RESERVED</td>
<td>GND</td>
</tr>
<tr>
<td>10</td>
<td>DATA CLK</td>
<td>RESERVED</td>
<td>D8</td>
</tr>
<tr>
<td>11</td>
<td>GND</td>
<td>RESERVED</td>
<td>D9</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>D10</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>D11</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>D12</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>GND</td>
<td>D13</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>GND</td>
<td>D14</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>GND</td>
<td>D15</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>W/R</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>GND</td>
<td>INTO*</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESERVED</td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>RESERVED</td>
<td>RESET</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>RESERVED</td>
<td>CLK2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>GND</td>
<td>A0</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>FLAGIN</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>-5V ANALOG</td>
<td>FLAGOUT</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>5V ANALOG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>AGND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>-15V</td>
<td>5V STBY</td>
<td>15V</td>
</tr>
<tr>
<td>32</td>
<td>15V</td>
<td>5V</td>
<td>5V</td>
</tr>
</tbody>
</table>

The asterisk (*) denotes an active-low signal.

Fig. 3.2 DSPLINK backplane interface connector
<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction with respect to DSP</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0-D15</td>
<td>to/from</td>
<td>Sixteen bi-directional TTL data lines of DSPLINK</td>
</tr>
<tr>
<td>GND</td>
<td></td>
<td>Digital ground</td>
</tr>
<tr>
<td>W*/R</td>
<td>from</td>
<td>DSPLINK read/write* line to signal the direction of data transfer</td>
</tr>
<tr>
<td>IOE*</td>
<td>from</td>
<td>An active-low, input/output enable signal indicating an access on the DSPLINK</td>
</tr>
<tr>
<td>INT0*</td>
<td>to</td>
<td>A negative-edge triggered, or active-low interrupt signal on DSPLINK generated on the GT processor I/F board.</td>
</tr>
<tr>
<td>RESET*</td>
<td>from</td>
<td>DSPLINK reset line.</td>
</tr>
<tr>
<td>CLK/2</td>
<td>from</td>
<td>General purpose clock signal. This signal is not used by the GT processor I/F board.</td>
</tr>
<tr>
<td>A0-A3</td>
<td>from</td>
<td>Four buffered TTL address lines of DSPLINK.</td>
</tr>
<tr>
<td>FLAGIN</td>
<td>to</td>
<td>General purpose input line on DSPLINK readable by the DSP.</td>
</tr>
<tr>
<td>FLAGOUT</td>
<td>from</td>
<td>General purpose output line on DSPLINK writeable by the DSP.</td>
</tr>
<tr>
<td>15V</td>
<td>-</td>
<td>15 volts power supply.</td>
</tr>
<tr>
<td>-15V</td>
<td>-</td>
<td>-15 volts power supply.</td>
</tr>
<tr>
<td>5V</td>
<td>-</td>
<td>5 volts power supply</td>
</tr>
<tr>
<td>-5V ANALOG</td>
<td>-</td>
<td>-5 volts analog power supply.</td>
</tr>
<tr>
<td>5V ANALOG</td>
<td>-</td>
<td>5 volts analog power supply.</td>
</tr>
<tr>
<td>AGND</td>
<td>-</td>
<td>Analog ground</td>
</tr>
<tr>
<td>5V STDBY</td>
<td>-</td>
<td>5 volts standby power supply.</td>
</tr>
<tr>
<td>HOP CLK</td>
<td>to</td>
<td>Hop clock signal originating from the GT processor I/F board.</td>
</tr>
<tr>
<td>HOP CLK*</td>
<td>to</td>
<td>Inverse hop clock signal. This pin is not currently being used by the GT processor I/F board.</td>
</tr>
<tr>
<td>DATA CLK</td>
<td>-</td>
<td>Data clock signal originating from the GT processor I/F board.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>-</td>
<td>Reserved lines for the DSP backplane.</td>
</tr>
</tbody>
</table>

The asterisk (*) denotes an active-low signal.

Table 3.1 DSP backplane interface pinout description

3.2.7.2 Serial Link Interfaces

There are two serial link interfaces required for the GT and payload simulators. The first interface is provided by a standard serial connector located on the host PC to access the synchronization response return serial link. As mentioned in Section 3.2.5, serial communications software routines were developed and written at DREO [10] to provide the user interface to the synchronization response return link.

The second serial link is used to relay the downlink synchronization reference pulse to the GT simulator. The downlink synchronization reference pulse serial link interface is a 9-pin D-type connector located on the DIB. The pinout description of the D-type connector is shown in Fig. 3.3.
Fig. 3.3 Pinout configuration for serial connector to downlink synchronization reference pulse

3.2.7.3 Data Source Interface

The connection between the data source and the DIB shown in Fig. 3.1 is realized by a male 9-pin D-type connector and is described further in [6].

3.2.7.4 HSC Command and Transmit Data Interface

As shown in Fig. 3.1, there are two connections between the GT processor i/f board and the HSC. These connections correspond to the HSC’s command and transmit data interface. The connections are made using a 26-pin ribbon cable and a 10-pin ribbon cable respectively. The details of the command and transmit data interface are described in [3] and [4].

3.2.7.5 Frequency Synthesizer Interface for the HSC

A 50-pin ribbon cable is used to connect the HSC to the frequency synthesizer. A description of the connection between HSC and the frequency synthesizer, as shown in Fig. 3.1, is given in [3].

3.3 Ground Terminal Simulator Software

The following section provides a description of the software for the DSP and the host. In both cases, the principal concepts and operations of the routines are outlined. The software for the DSP is described first. The DSP software includes the coarse and fine synchronization
routines and the processing of synchronization responses. Subsequently, the host/user interface software is described.

3.3.1 DSP Assembly Language Programs

The GT processor functions are implemented on a Texas Instruments' TMS320C30 DSP board which is contained in a PC. The software for the GT processor was written in assembly language and is described in the following subsections.

3.3.1.1 Main Assembly Language Program

The main DSP program for the GT processor performs two general functions. The first function is the preliminary initialization of the DSP board, the interface boards, and associated hardware of the GT simulator. The second function of the main DSP program is to respond to commands issued by the host program. The commands correspond to different modes of operation for the GT simulator. The two functions of the main DSP program are described further in the following subparagraphs.

3.3.1.1.1 Preliminary Initialization by the GT Processor DSP

The first stage of the main assembly program involves putting the DSP and its associated interface boards and hardware into a known state. The TMS320C30 DSP data page (DP) pointer, status register, stack pointer, primary bus control register, and secondary bus control register are initialized to appropriate values as specified in [11] and shown in Section B.3 of Appendix B. A software reset is then issued to the GT processor i/f board and DIB through the DSPLINK interface to reset circuits and latches. The software reset operations for the GT processor i/f board and the DIB are described in [4] and [6] respectively.

A subroutine is called next to download the GT parameters for the simulation. The parameters are contained in an ASCII data file which is read and processed by the host/user interface program. The parameters are then transferred from the host to the DSP through the dual port memory. A copy of the ASCII data file for the GT parameters is included in Section C.4 of Appendix C.

The next step in the initialization process is to set up the numerically-controlled oscillator (NCO) to start the GT clock circuit on the GT processor i/f board. The clock circuit generates the hop clock and data clock signals for the GT simulator. The NCO setup is described in [4]. The HSC is also initialized in this part of the main assembly language program. The HSC initialization is carried out in the same manner as for the GT parameters. An ASCII file containing the HSC parameters is read by the host/user interface program which then passes the values to the DSP board. The DSP board then loads the values onto the HSC through the GT processor i/f board [4].

18
The final step in the initialization process is to enable the interrupts for the DSP board. An interrupt is generated on the rising edge of the hop clock in order that appropriate counters for the GT processor be updated and synchronization probes be generated if applicable. The interrupt service routine for the GT processor is described further in Section 3.3.1.4.

3.3.1.1.2 GT Simulator Modes of Operation

Once the DSP board and associated hardware are initialized, the main assembly program enters into a loop to wait for and respond to user commands transferred to it by the host/user interface program. The commands relate to different modes of operation for the GT simulator as well as to terminating the simulation. When a command is received, the DSP processes it and determines the appropriate subroutine to execute. If a command is received to terminate the simulation, the DSP disables the interrupts and remains idle. The modes of operation are listed in Table 3.2. The first five modes are discussed further in Section 3.3.2. A description of the algorithms for the coarse and fine synchronization modes is provided in the next sections.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Transmitting a CW signal</td>
<td>GT simulator transmits one of five predefined continuous wave (CW) frequencies located in the frequency band of operation.</td>
</tr>
<tr>
<td>2</td>
<td>Run Mode</td>
<td>GT simulator switches the HSC to RUN mode [3] to randomly hop over the system bandwidth.</td>
</tr>
<tr>
<td>3</td>
<td>Sweep Mode</td>
<td>GT simulator sweeps a CW signal across the system bandwidth.</td>
</tr>
<tr>
<td>4</td>
<td>Frame Zero (FR0) Enable Mode</td>
<td>GT simulator monitors the downlink synchronization reference serial link for the downlink synchronization reference pulse and adjusts the GT hop clock counter if required.</td>
</tr>
<tr>
<td>5</td>
<td>FR0 Disable Mode</td>
<td>GT simulator ceases monitoring the downlink synchronization reference serial link.</td>
</tr>
<tr>
<td>6</td>
<td>Coarse Synchronization Mode</td>
<td>GT simulator performs coarse synchronization.</td>
</tr>
<tr>
<td>7</td>
<td>Fine Synchronization Mode</td>
<td>GT simulator performs fine synchronization.</td>
</tr>
</tbody>
</table>

Table 3.2 GT simulator modes of operation
3.3.1.2 Coarse Synchronization Assembly Routine

During coarse synchronization, the GT processor is responsible for several tasks. The tasks include generating coarse synchronization probes at appropriate times, processing synchronization responses transmitted from the payload simulator, and responding appropriately to these synchronization responses. When coarse synchronization is achieved, it is considered that the GT clock is within a hop from the payload system clock. The state diagram in Fig. 3.4 describes the general flow of events for the coarse synchronization process. The concepts or general procedure for the states shown in Fig. 3.4 are described in more detail in the following paragraphs. A listing of the assembly language program for the coarse synchronization routine is included in Section B.4, Appendix B.

Fig. 3.4 Coarse synchronization procedure state diagram

3.3.1.2.1 Starting the Coarse Synchronization Procedure

In this implementation of the GT simulator, a command is issued by the user to start the coarse synchronization procedure. The command is issued from the host/user interface program which is described in Section 3.3.2. When the coarse synchronization command is received, the GT processor initializes parameters related to the transmission of synchronization probes and the processing of synchronization responses from the payload simulator. First, the GT processor must calculate the start time for the HSC. As described in [3], the delayed start time is necessary
in order to give the HSC ample time for the propagation of the random number generator and to precompute hop frequencies for the start frame. The start time is calculated based on a frame boundary, i.e. the start time is given as hop 0 of a frame “X”. From [3], the earliest start time is calculated to be the third frame after the current frame. Where a terminal is not assigned all the coarse synchronization probe frames, the GT processor must wait until the first assigned frame following the delayed start time before transmitting the probes for the particular terminal. The start time is transferred to the HSC by way of the GT processor i/f board which is described in Section 3.2.2.2 and documented in [4].

When the start time is transferred to the HSC and the necessary parameters have been initialized for the coarse synchronization algorithm, the GT processor commands the HSC to switch to uplink synchronization mode and waits until the start time precalculation is completed before proceeding to state 2 to begin generating coarse synchronization probes.

3.3.1.2.2 Coarse Synchronization Probe Generation

The GT processor’s primary task in state 2 is to generate coarse synchronization probes at the appropriate time. In this implementation, only one terminal is realized to facilitate the demonstration of the coarse synchronization algorithm. In addition, the terminal allocation for transmitting coarse synchronization probes is selected to be on frames which are a multiple of four.

The process for achieving coarse synchronization consists of transmitting probes at different timing hypotheses. As described in Section 2.2.1, coarse synchronization probes are transmitted in the form of two bursts of sixteen probes each in the allocated channel and at the specific FSK tone bin. The coarse synchronization probes for the particular terminal implemented are to be transmitted on channel 1 and FSK tone bin 3. The two bursts of synchronization probes are transmitted with a timing offset between them. In this implementation, the second burst of synchronization probes is transmitted 1/2 hop later than the first burst of probes. If the received probes are detected by the payload processor, a “detect” response is formulated and relayed back to the GT simulator through the synchronization response return link. When the payload simulator is able to consistently detect the coarse synchronization probes for a timing hypothesis, it is considered that the timing of the GT clock is within a hop of the payload (system) clock.

The strategy selected to test the different timing hypotheses until the synchronization probes are detected involves starting at the most probable timing hypothesis and moving outward in increments of a hop at a time. The GT simulator uses the information obtained from the downlink synchronization pulse to derive the most probable timing hypothesis which does not account for any propagation delay. According to [2], the point at which synchronization probes are transmitted occurs at hop 288 of the time-frequency plan shown in Fig.2.3. As a result, the default starting point of the search for the correct timing hypothesis is selected to be at hop N=288 of a terminal’s assigned frame and corresponds to the case where there is perfect alignment of the GT clock with the payload clock.
Using the default starting point, the first series of coarse synchronization probes are transmitted starting on hop 288 of the terminal’s assigned frame. On the next iteration, the synchronization probes are delayed by a hop and are transmitted starting at hop (N+1)=289 of the terminal’s next assigned frame. The following series of synchronization probes are transmitted a hop earlier than the starting point, at hop (N-1)=287. With each iteration of the search, the timing hypotheses move further away from the starting point. The “outward moving” search scheme is illustrated in Fig. 3.5.

![Diagram of search scheme for coarse synchronization timing hypotheses](image)

**Fig. 3.5** Search scheme for coarse synchronization timing hypotheses

The timing hypotheses are stored in an array indexed by the frame number during which the synchronization probes were transmitted. This allows a hypothesis to be retrieved should it result in the synchronization probes being detected by the payload simulator. The actions following the detection of synchronization probes are discussed below.

A search range is included as a parameter for the search scheme and is used to limit how many timing hypotheses are tested in the search. The search range is given by the absolute value of the maximum offset from the starting point for which coarse synchronization probes will be transmitted. The search range is in units of hops and is user-configurable as a parameter which is downloaded from an ASCII data file during run time. The ASCII data file is discussed in Section 3.3.3. The default search range is set at 32 hops which results in the probes being transmitted with timing hypotheses of up to ±32 hops or ±1 subframe. If the coarse synchronization probes have not been detected when the entire search range has been exhausted, the search range is
exceeded and the GT processor proceeds to state 6 where an error is signalled to the user. State 6 is described in Section 3.3.1.2.6.

In addition to generating the coarse synchronization probes, the GT processor processes the synchronization responses which are returned by the payload simulator. As described in Section 2.2.3, the synchronization responses for each frame are transmitted to the GT simulator via the synchronization response return serial link. The synchronization response contains both coarse and fine synchronization estimates. During coarse synchronization, the GT processor processes only the coarse synchronization estimate corresponding to the terminal’s allocated probe frame. If no “detect” received in the appropriate coarse synchronization estimate, the GT processor does nothing further and continues with its task of generating coarse synchronization probes. However, if a “detect” is received in either of the bursts, the GT processor suspends the generation of synchronization probes for different timing hypotheses, and proceeds to state 3 to verify the “detect” received. The last timing hypothesis tested is saved before going to state 3 to facilitate resumption of the probe generation if the “detect” received proves to be an invalid one.

3.3.1.2.3 Verification of “Detect” Responses Received

In order to be certain that a “detect” response is valid, the GT processor retransmits synchronization probes using the timing hypothesis which prompted the “detect”. In this implementation, the GT processor uses the reference frame number included in the synchronization response to retrieve the appropriate hypothesis to be verified. The GT processor again waits until the terminal’s allocated probe frame to retransmit the synchronization probes. The GT processor examines the subsequent coarse synchronization estimates to confirm the detection of synchronization probes by the payload simulator. Therefore, just as in state 2, the GT processor also processes synchronization responses from the payload simulator. For this implementation, the user can define the number of iterations for synchronization probe retransmission to validate the “detect” response. As well, the number of subsequent “detects” which are to be received to confirm detection is user-configurable. The two configurable values are included in the same ASCII data file that contains the search range parameter mentioned in Section 3.3.1.2.2.

As shown in Fig. 2.4, there is a processing delay from the time the synchronization probes are transmitted to the time the corresponding synchronization response is received by the GT. As a result, when the GT processor first enters into state 3 to verify a “detect” response, the GT processor may continue to receive synchronization responses for probes sent prior to changing states. These synchronization responses are stored temporarily so that they may be processed later should a “false detect” result from the retransmitted synchronization probes during verification. A “false detect” occurs if the minimum number of “detects” required to confirm the detection for the retransmitted probes is not received for the number of iterations selected. If a “false detect” is concluded during the verification process, the GT processor enters into state 5 to process the temporarily stored synchronization responses. State 5 is described further in Section 3.3.1.2.5. In this implementation of the GT processor, only synchronization responses for the single terminal considered are stored in the synchronization responses buffer. As an added
precaution, the GT processor checks to see if there's any more room in the synchronization response buffer before storing the synchronization response. The overflow check serves as a debug feature to ensure the buffer is being cleared properly. If there is an overflow of the synchronization response buffer, then the GT processor goes to state 7 where a signal is sent to the user indicating the overflow error. The actions taken in state 7 are described further in Section 3.3.1.2.7.

If the verification is successful and the original “detect” is confirmed, the GT processor examines “detect” responses for the retransmitted probes to determine whether the GT clock should be delayed by 1/2 a hop. This additional step is required because as mentioned in Section 3.3.1.2.2, the two bursts of synchronization probes are transmitted with a timing offset of 1/2 hop between them. In essence, two timing hypotheses are transmitted in one probe frame during coarse synchronization. As a result, the synchronization response from the payload simulator consists of two estimates for each coarse synchronization channel as described in Section 2.2.3. Fig. 3.6 illustrates the possible scenarios of synchronization responses. It is noted that four hops per burst are used to simplify the illustration. As the second burst is delayed by 1/2 a hop, if the responses for the retransmitted probes indicates a majority of “detects” in the second burst, the GT clock is delayed by 1/2 hop. Once the adjustment is made the GT processor advances to State 4 which is described further in the following section.
(i) GT clock 1 hop earlier than payload clock

(ii) GT clock between 1/2 and 1 hop earlier than payload clock

(iii) GT clock 1/2 hop earlier than payload clock

Fig. 3.6 Different scenarios for GT clock alignment with payload clock
(iv) GT clock less than 1/2 hop earlier than payload clock

(v) GT clock aligned with payload clock

(vi) GT clock less than 1/2 hop later than payload clock

Fig. 3.6 (cont'd) Different scenarios for GT clock alignment with payload clock
(vii) GT clock 1/2 hop later than payload clock

(viii) GT clock between 1/2 and 1 hop later than payload clock

(ix) GT clock 1 hop later than payload clock

Fig. 3.6 (cont'd) Different scenarios for GT clock alignment with payload clock

3.3.1.2.4 Preparing for Fine Synchronization

In State 4, the GT processor uses the confirmed hypothesis to adjust the GT clock before proceeding to the fine synchronization algorithm. In this implementation, a signal is sent back to the user to indicate that coarse synchronization has been achieved. The user is then given the option to continue with fine synchronization or to halt the program. The user options are described in more detail in Section 3.3.2.
3.3.1.2.5 Clearing the Synchronization Response Buffer

A synchronization response buffer is used to store synchronization responses which arrive after the GT processor changes from state 2 (generating synchronization probes) to state 3 (verification of “detect”). The responses correspond to the synchronization probes transmitted after those which resulted in a “detect” response from the payload simulator. If the original “detect” response is shown to be invalid in state 3, the GT processor processes the stored responses to see if a “detect” was received for the subsequent synchronization probes. The buffer thus saves the GT processor from having to retransmit synchronization probes for the hypotheses tested. If a “detect” is received in a subsequent response, then the GT processor switches back to state 3 to verify whether this “detect” is valid or not. Otherwise, if all the stored responses are processed and no “detect” occurs, then the GT processor resumes generation of synchronization probes (state 2).

3.3.1.2.6 Search Range for Coarse Synchronization Routine Exceeded

When synchronization probes for all the timing hypotheses have been transmitted for the search range and no subsequent “detect” is received for any of the probes, a flag is set to alert the user. The user is then given a choice to repeat the downlink synchronization procedure or to halt the program. The user options are described in more detail in Section 3.3.2.

3.3.1.2.7 Synchronization Response Buffer Overflow

If an overflow occurs in the synchronization response buffer, then a flag is set to alert the user to the error. The host/user interface program for the GT simulator is subsequently aborted. The host/user interface program is described further in Section 3.3.2.

3.3.1.3 Fine Synchronization Assembly Routine

Once coarse synchronization is achieved, the GT performs fine synchronization in order to refine the GT clock alignment with the payload clock. For fine synchronization, the GT processor performs tasks similar to those during coarse synchronization. The GT processor generates fine synchronization probes to be transmitted at allocated times and processes synchronization responses transmitted by the payload simulator relating to the fine synchronization probes. During fine synchronization, the GT processor is also responsible for adjusting the GT clock according to the fine synchronization estimate received from the payload simulator. For this implementation, fine synchronization is considered achieved when the GT clock is within 10% of a hop from the payload clock. The choice of 10% was considered reasonable to account for frequency drift with minimal degradation to FSK modulation performance. The fine synchronization procedure is illustrated in Fig. 3.7. The general concepts of the states in Fig. 3.7 are described further in the following subsections. A listing of the assembly program for the fine synchronization routine is included in Section B.5, Appendix B.
Fig. 3.7 Fine synchronization procedure state diagram

3.3.1.3.1 Beginning Fine Synchronization

Upon receiving a command to perform fine synchronization, the fine synchronization routine begins by initializing variables associated with the transmission of fine synchronization probes. The DSP also sends a command to the HSC to go to RUN mode after which the HSC switches the frequency synthesizer according to a pseudorandom sequence [3]. The DSP then proceeds to state 2 to begin transmitting fine synchronization probes.

3.3.1.3.2 Generation of Fine Synchronization Probes

As in the case of coarse synchronization, only one terminal is realized to facilitate the demonstration of the fine synchronization algorithm. In addition, the terminal allocation for transmitting fine synchronization probes is again selected to be on frames which are a multiple of four.

The fine synchronization process involves transmitting fine synchronization probes during a terminal’s allocated frame and time slot. As described in Section 2.2.2, fine synchronization probes are transmitted in a single burst of 32 probes. Another difference between the two synchronization processes is in the way the probes are transmitted. For coarse synchronization, synchronization probe frequencies are precomputed and the GT processor sends commands to the HSC to transmit the probes. In fine synchronization, the frequency for the fine synchronization probe is transferred to the HSC on a hop basis. The fine synchronization probe frequency is defined by a channel and FSK tone bin. The channel and FSK tone bin information are passed to the HSC through the GT processor i/f board. The HSC then uses the information to
generate the appropriate transmit hop signal on the frequency synthesizer. In this implementation of the GT simulator, the fine synchronization probes are transmitted on channel 2, FSK tone bin 3. The GT processor's task is to determine when it is time to transmit the fine synchronization probes. The transmission of the fine synchronization probes is carried out by the interrupt service routine which is described in Section 3.3.1.4. When it is time to transmit fine synchronization probes, the GT processor sets a flag which is monitored by the interrupt service routine.

In addition to determining when fine synchronization probes are to be transmitted, the GT processor processes the synchronization responses which are returned by the payload simulator. Again, the synchronization responses for each frame are transmitted to the GT simulator on the synchronization response return serial link. The fine synchronization responses correspond to the payload simulator's estimate of how early or late the received probes are relative to the system clock. In this implementation, the host/user interface program of the GT simulator initially processes the fine synchronization responses and computes an average of ten responses received for the user. The host/user interface program also calculates a corresponding phase adjustment required for the NCO to refine the GT clock alignment. The average estimate of the responses and the phase change are then transferred to the GT processor. When the GT processor has read and stored the two values, the GT processor proceeds to state 3 to adjust the GT clock.

3.3.1.3.3 Refining the Alignment of the Ground Terminal Clock

Prior to adjusting the NCO phase to realign the GT clock, the GT processor checks whether the estimates received for the user are converging. The point of convergence is chosen to correspond to the GT clock being within 10% of a hop from the payload clock. In addition, the GT processor checks to see how many times the GT clock has been adjusted (or how many bursts of fine synchronization probes have been sent) to avoid the situation of endlessly trying to perform fine synchronization.

In this implementation, the adjustment of the GT clock is carried out over a frame (320 hops) to minimize frequency discontinuities while making the adjustment in a timely manner. Based on the estimate received from the synchronization response, a new GT clock frequency is calculated to align the GT clock over a period of 320 hops, afterwhich the original GT clock frequency is reinstated. The calculations for the new frequency which take into consideration this gradual approach is described in Section 3.3.2. If the synchronization response indicates that the probes received were early, the GT clock frequency is lowered to, in effect, delay the clock. Conversely, if the estimate shows that fine synchronization probes were late, then the GT clock frequency is increased to advance the clock. The clock frequency adjustment is implemented by adjusting the phase increment of the NCO which produces the clock signals for the GT simulator. In the assembly language program, a subroutine is called to transfer the new phase increment to the NCO via the GT processor i/f board. Once the new phase adjustment is loaded, the GT processor waits for 320 hops after which the original phase increment for the NCO is reloaded to produce the initial GT clock frequency.
3.3.1.3.4 Nonconvergence of Fine Synchronization Estimates

The arrival into this state implies that either the synchronization estimates being received from the payload simulator are not converging, or it has taken too long for the synchronization estimates to converge. At this point, the assembly language routine sets a flag to indicate to the host program that this event has occurred. The fine synchronization assembly language routine then returns to the main assembly language program to wait for the next command from the user.

3.3.1.3.5 Achieving Fine Synchronization

When fine synchronization is achieved, the GT processor sets a flag to the host program. At this point, the GT processor is ready to transmit user data. However, the development of the GT simulator was terminated here as the demonstration of synchronization concepts was the primary goal of this project.

3.3.1.4 Interrupt Service Routine

In the GT simulator implementation, an interrupt is generated on the rising edge of every hop clock pulse. When an interrupt is received by the DSP, the DSP stops its current task and invokes the interrupt service routine (ISR). The interrupt service routine carries out some routine updating of counters and performs tasks related to synchronization if applicable. Once the interrupt service routine has been executed, the DSP continues the task it was performing prior to receiving the interrupt. The interrupt service routine is written in assembly language and can be found in Section B.6, Appendix B. The tasks for the interrupt service are shown in Fig. 3.8 and are described in more detail in the following paragraphs.

3.3.1.4.1 ISR Housekeeping

The first step of the interrupt service routine consists of clearing the interrupt signal sent to the DSP by the GT processor i/f board. The interrupt signal is automatically cleared on the GT processor i/f board by reading the interrupt port of the GT processor i/f board [4]. The next step for the interrupt service routine is to update the hop and frame counters of the GT processor. Finally, the interrupt service routine checks to see if the GT processor is in either downlink synchronization, uplink coarse synchronization, or uplink fine synchronization modes. If the GT processor is not in one of these modes, the interrupt service routine is then complete and the DSP can resume its normal operations. Otherwise, the interrupt service routine performs one of the tasks described below before the DSP resumes its normal operations.
3.3.1.4.2 ISR Tasks for Coarse Synchronization Mode

For coarse synchronization, the interrupt service is responsible for transmitting the coarse synchronization probes at the appropriate time. In order to determine the appropriate time for transmitting the coarse synchronization probes, the interrupt service routine goes through a series of checks. As described in [3], when the HSC is changed to CSYNC (coarse synchronization) mode, the HSC first requires a certain amount of time to precompute the hop frequencies for the coarse synchronization probes. Thus, the interrupt service routine first verifies that the precomputation time has elapsed. After the precomputation time has elapsed, the interrupt service routine must check to see if the current hop is hop 0. During coarse synchronization, the HSC must be given an FCALC (frequency calculate) command on hop 0 of every frame in order to calculate the next synchronization probe frequencies. The next step is to verify whether the current frame is one that is assigned to the user. If it is an assigned frame, the interrupt service routine checks to see if it is time to transmit coarse synchronization probes based on a particular timing hypothesis. Once it is the appropriate time, the interrupt service routine must differentiate between transmitting one of the two bursts of probes with a timing offset of 1/2 hop between them as described in Section 3.3.1.2.2. The transmission of the coarse synchronization probes is
accomplished by sending ULGO (uplink GO) commands to the HSC through the GT processor i/f board [3, 4].

3.3.1.4.3 ISR Tasks for Fine Synchronization Mode

As described in Section 2.2.2, during fine synchronization, the GT processor transmits a series of bursts of 32 fine synchronization probes at specific times allocated to a particular terminal. As a result, the interrupt service routine must perform a similar series of checks as for coarse synchronization in order to determine when to transmit the synchronization probes. First, the interrupt service routine checks to see if the current frame is an allocated frame for fine synchronization. If so, the interrupt service routine checks for the appropriate time to transmit the fine synchronization probes. The fine synchronization probes are transmitted by writing the channel and FSK bin information to the FSK/FRAME port of the GT processor i/f board [3, 4]. If the current frame is not allocated for the transmission of fine synchronization probes, a TXOFF (transmit off) command is written to the FSK/FRAME port which causes the transmit signal to be attenuated, effectively turning the transmitter off.

3.3.1.4.4 ISR Tasks for Downlink Synchronization Mode

As described in Section 2.1, downlink synchronization is performed in order to acquire the satellite downlink. In practice, downlink synchronization consists of the ground terminal detecting synchronization aids transmitted by the payload. The detection of the synchronization aids also provides the ground terminal with some preliminary information to begin uplink synchronization. A simulated downlink synchronization link was implemented whereby a reference pulse is transmitted on a serial link connecting the payload and GT simulators. The rising and falling edges of the reference pulse were chosen to correspond to the start of hop0/frame0 and hop0/frame 1 respectively. Access to the reference pulse is achieved by way of the status register on the DIB [6]. When the ground terminal processor is in downlink synchronization mode, the interrupt service routine reads the status register of the DIB and examines whether a rising or falling edge has occurred. If either edge is detected, the interrupt service routine verifies the hop and frame counters are set properly or resets the counters as required.
3.3.2 Host/User Interface Program

The user interface to the GT simulator is provided by the host/user interface program which is written in C. A listing of the host/user interface program is included in Section B.2, Appendix B. The host/user interface program initializes and downloads the DSP code to the DSP board (also referred to as the GT processor) and allows the user to select the mode of operation for the GT simulator. Furthermore, the host/user interface program monitors the status of the simulation by continuously checking a series of flags and responding as required. The general flow of the program steps is shown in Fig. 3.9. A description of each of these steps or tasks is provided in the following paragraphs.

![Host/user interface flow diagram](image)

Fig. 3.9 Host/user interface flow diagram

3.3.2.1 DSP Board Initialization

The first step of the host/user interface program involves using interface library subroutines provided by Spectrum Signal Processing Inc. [11] to properly initialize the DSP board which operates as the GT processor. Once the appropriate DSP board is selected and initialized, the DSP code is downloaded into the DSP program memory. The dual port memory of the DSP board, which allows for data transfer between the DSP and host PC and vice versa, is then initialized. Finally, a reset is issued to the DSP board using the interface library functions to start the execution of the DSP program.

3.3.2.2 Downloading Simulation Parameters

In order to allow flexibility in changing parameter values for a particular simulation, parameters are stored in ASCII files which are read by the host/user interface program and subsequently transferred to the DSP board. For this implementation of the GT simulator, there are three ASCII data files. The three data files include values for general GT parameters, for the HSC, and for CW frequency values. The data files are described further in Section 3.3.3 and are included in Appendix C. In the host/user interface program, each value is read and then formatted to be downloaded to the DSP. The downloading of the parameter values is also reflected in the DSP main assembly program which is discussed in Section 3.3.1.1.1.
3.3.2.3 Serial Communications Initialization

As described in Section 3.1, the experimental setup consists of the ground-based payload and GT simulators being located approximately 1.5 km apart. In order to facilitate integration and testing of the software, a remote operation capability was implemented for the GT simulator. A serial communications link was installed to connect the two simulators. Using the serial communications software developed at DREO [10], the GT simulator can be operated by the payload simulator. It is noted that the scope of the remote operation only encompasses the mode of operation for the GT simulator. The GT simulator still has to be powered on and the executable file has to be run locally before remote operation can occur. Similarly, if a catastrophic error occurs and a reboot of the GT simulator is required, the reboot must also be performed at the local terminal.

3.3.2.4 User Interface Menu

During the development of the GT simulator, several modes of operation were implemented. The modes of operation were introduced in Section 3.3.1.1.2. While some of these modes are not part of the uplink synchronization process, they are retained for debug purposes. A mode is selected by the user from a menu which is displayed on the local terminal. A replica of the user menu which is displayed on the local screen is shown in Fig. 3.10. Each of the options in the menu are described further in the following paragraphs.

```
* ***************************************************** *
*                  CW test - 7-11 April 1997              *
*         GT Synch Processor Menu                       *
* ***************************************************** *
* Enter one of the following:                           *
* 'L' : Go to lower edge of hop BW                     *
* 'U' : Go to upper edge of hop BW                     *
* 'M' : Go to middle of hop BW                         *
* 'Q' : Go to one quarter mark of hop BW               *
* 'T' : Go to three quarter mark of hop BW             *
* 'F' : Go to specific frequency                       *
* 'R' : Go to RUN mode                                 *
* 'S' : Slowly cycle through hop BW                    *
* 'E' : Enable interrupt/FR0 detection                 *
* 'D' : Disable interrupt/FR0 detection                *
* 'C' : Coarse Synchronization test                    *
* 'W' : Fine Synchronization test                      *
* 'X' : Exit program or stop slow hopping (option 'S') *
* TMS interrupts/FR0 detection is currently disabled.  *
* ***************************************************** *

Enter selection:
```

Fig. 3.10 User menu for the GT simulator
The first six options on the user menu allow the user to select a CW frequency tone to be transmitted. The first five of six options ('L', 'U', 'M', 'Q', 'T') correspond to the lower edge, the upper edge, the middle, the quarter mark, and the three-quarter mark of the hopping bandwidth respectively. These values are stored in an ASCII data file which is read by the host/user program as described above. The sixth option ('F') corresponds to an arbitrary value within the hopping bandwidth, selectable by the user.

The RUN mode option ('R') causes the HSC to randomly hop over the hopping or system bandwidth. Alternatively, the 'S' option transmits a CW tone that is swept across the hopping bandwidth. The user can select the rate at which the tone is swept through the bandwidth by entering a dwell time (in seconds) and a hopping increment (between 1 and 16777215) for the prompts following the 'S' entry.

The 'E' option enables the detection of the downlink synchronization reference pulse (FR0 pulse). When this option is invoked, the DSP returns a flag every time the rising edge of the FR0 pulse is detected. The host/user interface program prints a message on the local screen indicating that the rising edge of the FR0 pulse was detected and prints the values of the hop and frame counters when the edge is detected. The values indicate whether an adjustment is required for the counters. When no adjustments are observed for the counters over a period of time, downlink synchronization can be considered to be achieved and thus, the detection of the FR0 pulse can be disabled. To disable this option, a 'D' may be entered at any time during this mode of operation.

The 'C' and 'W' allow the user to execute the coarse and fine synchronization algorithms for the GT simulator, respectively. The detailed description of the algorithms are included in Sections 3.3.1.2 and 3.3.1.3.

3.3.2.5 Host/User Interface Loop

Once the DSP and host initialization have taken place and the user menu is displayed, the host enters into a loop operation to respond to user commands and to flag conditions relating to the status of the simulation. Fig. 3.11 illustrates the loop. The order in which the loop is carried out is only a reflection of the order in which the GT processor functions were developed.
3.3.2.5.1 Checking for User Input - Local Keyboard Input

If the host detects a keyboard entry, it will process the input to determine whether the input is valid and if so, take appropriate action. The valid entries are the options on the user menu described in Section 3.3.2.4. The subroutine for processing keyboard entries has been written to be case insensitive (i.e. the host program interprets upper and lower case entries to be the same). If a valid entry is received, the host transfers the appropriate command to the DSP board. In addition, if serial communications between the payload and GT simulators is activated, a keyboard entry at the local terminal will also cause a message to be sent to the remote terminal identifying the entry made. If an invalid entry is received, the host program prints an error message, prompts the user for another entry and displays the user menu again.

3.3.2.5.2 Checking for User Input - Remote Input

The host program also checks for remote input if serial communications is enabled between the payload and GT simulators. Similar to any keyboard input, the host program processes the remote input to determine whether it is valid. The input options for remote input correspond to the options on the user menu. Remote input through the serial communications
link is formatted as text messages which must be read and decoded by the host program. If a valid entry is received, the host transfers the appropriate command to the DSP board and prints a message on the local terminal that a remote entry has been made.

3.3.2.5.3 Checking for FR0 Pulse Edge Detection

The task of checking for the FR0 pulse edge detection was originally used in the development of the downlink synchronization mode for the GT simulator. The task has been retained for debug purposes. The task involves reading a location in dual port memory to see if the flag for an FR0 pulse edge detection is set. If the flag is set, the host program clears the flag and displays a message on the screen including the value of the hop and frame counters before the edge was detected. The detection of the FR0 pulse can be disabled at any time by entering ‘D’.

3.3.2.5.4 Checking for Completion of Task by DSP

In this step of the loop operation, the host program checks to see whether the DSP is ready to receive another user command by reading the C30DONE flag which is also stored in dual port memory. When the C30DONE flag is set, the host program prompts the user for another command.

3.3.2.5.5 Checking for Return Link Synchronization Response

Synchronization responses are received by the GT simulator from the payload through a serial communications link and stored in a buffer on the host PC. When a synchronization response is received, the host program reads and decodes the response according to the format presented in Section 2.2.3. If the GT simulator is in coarse synchronization mode, the response is transferred to the DSP. However, in the case of fine synchronization, an average of ten fine synchronization responses is computed first. An average value of the fine synchronization estimates was chosen over a single estimate to provide further accuracy of the realignment required for the GT clock and possibly reduce the number of adjustments needed to align the GT clock with the payload clock. The average is then used to compute the phase change required on the NCO to align the GT clock. Both the average fine synchronization response and the corresponding phase change required are then transferred to the DSP.

3.3.2.5.6 Checking Other Flag Conditions

The final step of the loop operation for the host program is to check a series of flags which indicate the status of the simulation. Again, while some flags may not explicitly be part of the uplink synchronization process, they are retained for debug purposes. Table 3.3 describes each of the flags which are monitored by the host program and any subsequent options should a
flag become set. Again, the order in which the flags are listed only reflect the order in which the functionality was added in the development of the GT simulator.

3.3.3 ASCII Data Files

To facilitate the reconfiguration of parameters for the GT processor and the HSC during the trials without the need to recompile code, three ASCII data files are used. These data files must be stored in the same directory as the executable file for the host/user interface C program. The first data file called “freq.dat” contains the frequency values at specific points within the hopping bandwidth. The frequency values are used to generated CW tones to verify the system integration of the GT and payload simulators as well as to verify the SATCOM link. Frequency values are included for both the COMSTRON and SCITEQ frequency synthesizers which are supported in the uplink synchronization trials. The second data file, “hscinit.dat”, contains the parameters to be initialized on the HSC. Again, initial values are included for both the COMSTRON and SCITEQ synthesizers. The third data file called “GTparam.dat” allows the user to select values for GT processor variables. The user-definable variables enable changes to the coarse synchronization and fine synchronization procedures. These include the search limit for the synchronization probe hypotheses, the number of times a “detect” must be received to confirm coarse synchronization, the synchronization response buffer size, and the range of fine synchronization estimates within which fine synchronization is considered to be achieved. The ASCII files are included in Appendix C.
<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNG_XCDED</td>
<td>Indicates that the search range for the coarse synchronization algorithm has been exceeded with no coarse synchronization probes being detected. The search range is defined by the user in an ASCII data file which is read by the host program. This flag is used to indicate that the range of timing hypotheses to be tested has been exceeded. The occurrence of this flag being set may indicate the possibility of an error with the downlink synchronization acquisition. Thus, in the event this flag is set, the user is given the option to return to downlink synchronization mode.</td>
</tr>
<tr>
<td>CSYNC_OK</td>
<td>Indicates that coarse synchronization has been achieved. At this point, the detection of coarse synchronization probes for a particular timing hypothesis has been confirmed. The timing hypothesis is used to adjust the hop and frame counters (i.e. GT clock). If this flag becomes set, the user is given the option to go to fine synchronization mode.</td>
</tr>
<tr>
<td>PLINE_FLAG</td>
<td>Indicates that the buffer used by the GT processor to store coarse synchronization responses is full. The buffer is used when the GT processor is verifying a timing hypothesis which resulted in a “detect” response from the payload. The buffer holds only responses for timing hypotheses which were transmitted after the timing hypothesis which resulted in the “detect” response but before the verification procedure commenced. The verification of timing hypotheses is described in Section 3.3.1.2.3. This flag is used for debug purposes.</td>
</tr>
<tr>
<td>TOO_MANY_HYPS</td>
<td>Indicates that the buffer used by the GT processor to store the reference frame number and timing hypotheses is full. This condition occurs when the GT processor tries to overwrite another entry. This flag is used for debug purposes.</td>
</tr>
<tr>
<td>FRM_NOT_FOUND</td>
<td>When a coarse synchronization response is received from the payload, it is only referenced by the frame number to which the response corresponds. Thus, the GT processor must retrieve the timing hypothesis used in that particular frame to verify the hypothesis. The GT processor retrieves the timing hypothesis from the hypothesis log. If the frame number is not found and the timing hypothesis cannot be retrieved, the FRM_NOT_FOUND flag is set. The host program is subsequently aborted. This flag is used for debug purposes.</td>
</tr>
<tr>
<td>FSTART_AVAIL</td>
<td>Indicates that the frame number of the first valid fine synchronization response has been transferred by the GT processor. The frame number is stored by the host program and used to start processing fine synchronization responses.</td>
</tr>
<tr>
<td>FSYNC_OK</td>
<td>Indicates that fine synchronization has been achieved. A message is printed on the screen that the GT is ready to transmit data. Due to time constraints, the data transmission function has not been implemented on the GT simulator.</td>
</tr>
<tr>
<td>NO_FSYNC</td>
<td>Indicates that fine synchronization could not be achieved as the fine synchronization responses did not converge. If this condition occurs, the user is given the option to go back to coarse synchronization.</td>
</tr>
<tr>
<td>UFLO_CDTN</td>
<td>Indicates that data was not transferred to the GT processor if board in time for the next hop during fine synchronization, resulting in an underflow condition. A message is printed on the local terminal screen and the host program is subsequently aborted.</td>
</tr>
</tbody>
</table>

Table 3.3 Description of flags for GT simulator
4.0 Summary

This report describes the development of a GT simulator for an in-house activity examining synchronization aspects of EHF SATCOM at DREO and CRC. The GT simulator consists of a GT processor, a number of custom interface boards, frequency synthesizer, RF equipment, and a data source. The GT processor was implemented on a TMS320C30 DSP board from Spectrum Signal Processing Inc., and is contained in a host PC. A GT processor i/f board was designed and fabricated to generate the necessary clock signals for the GT processor and to provide the interface between the GT processor and an HSC. The HSC, in turn, controlled a frequency synthesizer used to produce the transmit signal. A multipurpose data interface board was also designed and fabricated to provide the interface between the GT processor and a data source, and to provide the interface to the downlink synchronization reference link.

The modes of operation for the GT simulator include: transmitting CW tones at specific points in the hopping bandwidth; transmitting an arbitrary CW tone within the hopping bandwidth; sweeping a CW tone across the hopping bandwidth; performing downlink synchronization using a simulated downlink; performing uplink coarse synchronization; and performing uplink fine synchronization. The first three functions were implemented during the development of the simulator and are retained for debug purposes.

The GT simulator was developed for use in uplink synchronization trials over the UK Skynet 4A satellite. The synchronization procedure begins with downlink synchronization. In this implementation, a simulated downlink is setup between the payload and GT simulators via an RS232 serial communications link. A downlink synchronization reference pulse is continuously transmitted by the payload simulator to the GT simulator. The edges of the reference pulse correspond to specific instances in the pseudorandom hopping sequence of the payload simulator. The GT simulator detects and uses this reference pulse to begin uplink synchronization.

There are two stages in performing uplink synchronization: coarse and fine synchronization. Both involve the transmission of synchronization probes to the payload simulator at allocated times. Synchronization responses to the synchronization probes are formulated by the payload and returned to the GT simulator. In coarse synchronization, a binary “detect/no detect” response is returned. In fine synchronization, a time estimate of how early or late the received probes are relative to the system clock is provided on the synchronization response return link. The synchronization response return link is implemented on a serial communications link capable of data transfer at 9.6 kb/s. These responses are transmitted to the GT simulator once every frame.

For coarse synchronization, two consecutive bursts of sixteen synchronization probes are transmitted at different timing hypotheses. In this implementation of the GT simulator, an “outward moving” search scheme is used in coarse synchronization to test different timing hypotheses beginning with the most probable hypothesis. In addition, a verification process is
carried out when a "detect" response is first received by the GT simulator for a particular timing hypothesis. Coarse synchronization is considered achieved when the GT clock is aligned to within a hop of the payload clock.

In fine synchronization, a single burst of thirty-two synchronization probes is transmitted. In order to reduce the number of times the GT clock is adjusted and to minimize any estimate errors due to noise, an average of ten fine synchronization responses is used to determine the fine adjustment required by the GT clock during fine synchronization. Fine synchronization is considered achieved when the GT clock is aligned to within 10% of a hop of the payload clock.

The GT simulator has been developed so that it can be remotely operated by the payload simulator once the GT simulator is powered on and the executable file is run. The remote operation capability is included to facilitate the operation of the trials since the GT and payload simulators are physically located 1.5 km apart.

Data files containing parameters for the simulation are used to allow changing of their values without recompilation. The parameters are downloaded by the host PC to the DSP at the beginning of the simulation.

A user's guide is included in Appendix A for installation and operating procedures of the GT simulator. Appendix B contains a listing of the software programs used by the GT simulator. Appendix C contains a listing of the parameter data files used by the GT simulator.
References


Appendix A: Ground Terminal Simulator Installation Guide

A1 Installation

A1.1 Hardware installation

The GT simulator (excluding the RF components) consists of the following:

- IBM-AT compatible PC
- Spectrum Signal Processing Inc., TMS320C30 DSP Board (GT processor)
- GT processor i/f board
- Hopping synthesizer controller
- Frequency synthesizer (COMSTRON or SCITEQ)
- Multipurpose DIB
- Data source (HP1645 Bit Error Rate Analyser)
- DSPLINK backplane cage

The DSP board or GT processor is installed in an ISA expansion slot of the PC. The configuration and jumper settings of the DSP board are described in [11]. For the uplink synchronization experiments, the default settings for the DSP board are used. The GT processor i/f board and the multipurpose DIB are each inserted into a slot of the 96-pin DSPLINK backplane. The DSPLINK backplane was described in Section 3.2.7.1 of the main document. Details of the configuration of the GT processor i/f board and DIB are provided in [4] and [6] respectively. Communications between the GT processor and the interface boards are made possible through the DSPLINK interface of the DSP board [11] using a 50-pin ribbon cable and DSPLINK extender card. The extender card, which is inserted one of the other slots of the DSPLINK backplane cage, maps the fifty lines of DSPLINK to the 96-line backplane.

On the other end of the GT processor i/f board, there are two connections for the HSC: the HSC command interface; and the HSC data interface connection. A 26-pin ribbon cable and a 10-pin ribbon cable are used respectively to connect the GT processor i/f board to the corresponding connectors on the back of the HSC box. An additional 50-pin ribbon cable connector is located on the back of the HSC box and is used to connect the HSC to the frequency synthesizer. Details of the HSC configuration and interfaces are found in [3].

On the other end of the DIB, there are three connectors. A 26-pin ribbon cable connector is available to access one of the debug latches on the DIB. The second connector is a 9-pin male RS232 type connector. It is used to connect the data source (HP1645) to the DIB. Another 9-pin female RS232 type connector is used for the downlink synchronization reference serial communications link. Full details of the DIB connectors are found in [6].
The serial port of the PC is also used in the GT simulator. The serial port of the PC is connected to the synchronization response return serial link.

A1.2 Program files

In order to run the GT simulator, several files must be present. The files are listed below:

\[
\begin{align*}
&\text{tms\_ul2\_out} \\
&\text{ulsync2\_exe} \\
&\text{com\_h}
\end{align*}
\]

The GT simulator programs are compiled and executed in DOS. The \texttt{tms\_ul2\_out} file is the output file from the TMS320C30 linker. The TMS320C30 linker creates the executable module by combining object files of compiled assembly programs and allocates sections to the DSP memory map. The GT simulator assembly routines being linked are shown in Fig. A.1. The file \texttt{rts\_lib} is the run time library routines which are supplied by Spectrum Signal Processing Inc. The options and filenames for the linker are contained in a command file which in included in Section B7, of Appendix B.

![Flowchart of TMS320C30 linker process](image)

**Fig. A.1 GT simulator TMS320C30 linker process**

Similarly, the \texttt{ulsync2\_exe} file is the output of the C linker for the PC. The C linker combines the compiled host/user interface program with the TMS320C30 development library routines, \texttt{lm30dev\_lib}, as well as the compiled serial communications routines. The definitions for the routines are found in the \texttt{com\_h} header file. The result of the compiled routines is contained in two object files, \texttt{com\_obj} and \texttt{serial\_obj}. Fig. A.2 illustrates the flow of the C linker process for the PC.
Fig. A.2 GT simulator host PC linker process
Appendix B: Software Listings

B1. GT simulator host/user interface program

/* Program Name: ULSYNC2.C */
/* Author: C. Tom */
/* Date Edited: 31 March 1998 */
/* THIS VERSION HAS A USER SELECTABLE OPTION FOR THE SERIAL COMMUNICATIONS */
/* ROUTINES. USER IS PROMPTED AT THE BEGINNING OF PROGRAM. */
/* */
/* Description */
/* */
/* ********** */
/* */
/* PC program for GT simulator for uplink synchronization trials. Includes */
/* capability for remote operation through serial communications link */
/* software. */
/* */
/* Application */
/* */
/* * 1. Available options to send CW at low, high, mid. 1/4, 3/4 band */
/* * 2. Available option to send CW at user specified frequency */
/* * 3. Available option to sweep through allocated BM (FSK seq) */
/* * 4. Available option to go to RUN node (frequency hopping) */
/* * 5. Available option to enable/disable FRAME 0 detect (confirm DL) */
/* * and adjust hop and frame counters. */
/* * 6. Available option to test CSYNC procedure */
/* * 7. Available option to test FSYNC procedure */
/* */

****** Include files ******
#include <stdio.h>
#include <conio.h>
#include "com.h"

****** THIS DPMEM address definitions ******
#define BASEID 0x290
#define BASEDP 0x307000L
#define CIODONE BASEDP + 6L
#define MODE_AVAL BASEDP + 3L
#define MODE BASEDP + 2L
#define MODE_ACK BASEDP + 1L
#define F_AVAIL BASEDP + 0L
#define F_VALUE BASEDP + 7L
#define F_ACK BASEDP + 6L
#define MSG_AVAIL BASEDP + 5L
#define MSG_NAMED BASEDP + 4L
#define MSG_ACK BASEDP + 3L
#define Dwell_AVAIL BASEDP + 2L
#define Dwell_VAL BASEDP + 1L
#define Dwell_ACK BASEDP + 0L
#define VAL_XDI BASEDP + 7L
#define VAL_ID BASEDP + 6L
#define VALUE BASEDP + 5L
#define VAL_ACK BASEDP + 4L
#define END_ASCI BASEDP + 3L
#define STOP_NOD BASEDP + 2L
#define INCR_AVA BASEDP + 1L
#define INCR_XS BASEDP + 0L
#define STOP_INT BASEDP + 7L
#define RIS_DET BASEDP + 6L
#define HFP_AVAIL BASEDP + 5L
#define USER_HFP BASEDP + 4L
#define HFP_ACK BASEDP + 3L
#define DBUG_AVAIL BASEDP + 2L
#define DBUG_VALUE BASEDP + 1L
#define DBUG_ACK BASEDP + 0L
#define CHK_AVAIL BASEDP + 7L
#define CHK.VALUE BASEDP + 6L
#define CHK_ACK BASEDP + 5L
#define FPM_REF BASEDP + 4L
#define CO_RESP BASEDP + 3L
#define CI_RESP BASEDP + 2L
#define F0_RESP BASEDP + 1L
#define F1_RESP BASEDP + 0L
#define RESP_AVAIL BASEDP + 7L
#define RESP_ACK BASEDP + 6L
#define RSC_XSDE BASEDP + 5L

B1
/* Define constants */

#define CSYNC_OK BASED + 0x28L
#define CHK_HOP BASED + 0x22L
#define CHK_FRM BASED + 0x22L
#define LOG_END BASED + 0x2EL
#define PLINE_FLAG BASED + 0x2FL
#define DAT_AVAIL BASED + 0x30L
#define INDEX BASED + 0x31L
#define DAT_VALUE BASED + 0x32L
#define DAT_ACK BASED + 0x33L
#define END_FILE BASED + 0x34L
#define TOC_MANY_HYPS BASED + 0x35L
#define FRM_NOT_FOUND BASED + 0x36L
#define FSTART_AVAIL BASED + 0x40L
#define FSTART_FRM BASED + 0x41L
#define FSTART_ACK BASED + 0x42L
#define EST_AVAIL BASED + 0x43L
#define FINE_EST BASED + 0x44L
#define PHS_CHANGE BASED + 0x45L
#define EST_ACK BASED + 0x46L
#define NO_FSYNC BASED + 0x47L
#define FSYNC_OK BASED + 0x48L
#define UPLC_CDN BASED + 0x49L

/***** Miscellaneous constants *****/

#define DUAL 0 /* Memory type, p.48 in manual*/
#define ALL 1 /* Memory type, p.48 in manual*/
#define BITCLR 0L
#define BITSET 1L
#define maxlen 80
#define local 0
#define remote 1

#define MAX_HOP_NUM 0xffffffff /* 16777215 for PN seq generator in HSC */
#define NCO_VAL 0x13A92A3L /* value of phs_mde for NCO - default 192000 Hz*/

#define RESP_BUFF_SZ 100 /* Synch response buffer size */
#define STRING_LEN 220 /* Serial com message length */

/"* External variable declaration */
/*****************************/

int i, err, checkpoint;
int ser_select = 0;
unsigned long check;
unsigned long io_band, up_band, mid_band, oneq_band, threeq_band;
unsigned long base_f, hop_bw;
unsigned long flag, flag1;
int count, mode_of_op;
int stop_fsel;
unsigned long user_f, freq_val;
double hop_par;
float dwell_time;
unsigned long hop_incr;
unsigned long dwell_cycles;
FILE *fp, *f2p, *f3p, *f4p, *f5p;
char fsting[maxlen], linetype, go_on_flg;
unsigned long idx;
long int data;
int int_mode = BITCLR;
int stop_pc = BITCLR;
int key_ret = BITCLR;
int msg_ret = BITCLR;
int proc_ret = BITCLR;
int operation;
int mlocal, mdest;
char string[STRING_LEN];
long int hypothesis;
long int sync_iter;
long int chk_point;
long int hop_bredge, frm_bredge;
long int command_given = BITCLR;
char c_resp_arr[RESP_BUFF_SZ][STRING_LEN];
long int f_resp_arr[RESP_BUFF_SZ];
int srcmp_ret = BITCLR;
long int resp_rd_ptr = 0;
long int resp_wr_ptr = 0;
long int frame, c0_crsrp, c1_crsrp, c0_fram, c1_fram;
long int datag_entry;
unsigned long frm_ref_4_fine;
long int cum_fine_resp = 0;
int fine_cnt = 0;
long int f_est_avg, phs_change_4_nco;
float adjustment, fit_avg;
char u_input;

/"* Subroutine declarations */
void Init_DPMMEM(void); /*initializes TMS dual port memory*/
unsigned long CTGet32Bit(int baseadr, long loc); /*alternate Get32Bit for DSP*/
int checkkey(int mdset); /*check and action key presses*/
int checkmsg(void); /*check for receive messages and others*/
void pabort(char *msg); /*print message, close file, exit*/
void Disp_menu(void);
void send_mode(int mode_sel);
void send_f(unsigned long _f_val);
int proc_msg(int type, char mcontent[220]);
float Rd_value(char par[10]);

****
main()
****
{

******Load c30 program******
check = SelectBoard(BASEIO); /*Initialize c30*/
printf("\nReturn from SelectBoard = 0x %x h\n", check);
err = LoadObjectFile("tms_u12.out"); /*Load c30 program*/
switch (err) {
case 0:
    printf ("\nTMS program has been loaded successfully.\n");
    break;
case 1:
    printf ("\nERROR. Unable to open TMS file.\n");
    exit(1);
    break;
case 2:
    printf ("\nERROR. Invalid address for TMS program.\n");
    exit(1);
    break;
default:
    printf("\nError. Value = %i", err);
    exit(1);
} /*end switch (err)*/

******Initialize TMS DP memory and synch response buffer in PC******
Init_DPMMEM();
for (i=0; i < RESP_BUF_SIZE; i++)
    { strcpy(&c_resp_arr[i][0], "empty");
      f_resp_arr[i] = 999;
    } /*end for i*/

******Start c30******
printf("\nSerial Comms routines enabled, enter '1', otherwise enter '0': ");
scanf("%i", &ser_select);
printf("\nSer_select value: ti ", ser_select);
Reset();
printf("\nReset issued to c30 board.\n"); /*Reset DSP board*/

******Download GT processor parameters******
if ((fp = fopen("GTparam.dat", "r")) == NULL) 
    { printf("\nError. Cannot open file GTPARAM.DAT. Aborting.\n");
      exit(1);
    } /*end if fp*/
printf("\nFile opened for GT parameters.\n");
while (fgets(fstrng, maxlen, fp) != NULL) 
    { printf("\nReading next GT parameter,\n"); scanf(fstrng, " %i \%i \%i \%i \%i \%i \x", &linetype, &idx, &data);
        if (linetype == 'd') 
            { check = Put32Bit(INDEX, DUAL, idx);
                check = Put32Bit(DAT_VALUE, DUAL, data);
                check = Put32Bit(DAT_AVAL, DUAL, BITSET);
                flag = BITCLR;
                while (((flag = CTGet32Bit(BASEIO, DAT_ACK)) != BITSET));
                check = Put32Bit(DAT_ACK, DUAL, BITCLR);
            } /*end if linetype*/
} } /*end while fgets*/
printf("\nEnd of transferring GT parameters.\n");
check = Put32Bit(END_FILE, DUAL, BITSET);

/***** Send NCO initial condition to TMS******/
check = Put32Bit(PHS_RDDED, DUAL, NCO.VAL);
check = Put32Bit(PHS_AVAI, DUAL, BITSET);
flagl = BITCLR;
while (((flagl = CGet32Bit(BASEIO, PHS_ACK)) != BITSET))
    printf("\nWaiting for TMS to ack transfer of NCO value. \n");
check = Put32Bit(PHS_ACK, DUAL, BITCLR);

/***** Read ASCII file with HSC parameters and transfer to TMS******/
printf("\nReady to read HSC parameters.\n");
if ((fip=fopen("hascin.txt", "r")) == NULL) {
    printf("\nError. Cannot open input file for HSC parameters.\n");
    exit(1);
} /*end if fip*/
printf("\nInput file opened for HSC parameters.\n");
while (fgets(fstring, maxlen, fip) != NULL)
{
    printf("\nReading HSC parameters into TMS memory.\n");
    sscanf(fstring, "%c %i %lx", &linetype, &idx, &data);
    if (linetype == 'd')
    {
        check = Put32Bit(VAL_IDX, DUAL, idx);
        check = Put32Bit(VALUE, DUAL, data);
        check = Put32Bit(VAL_RDY, DUAL, BITSET);
        flagl = BITCLR;
        while ((flagl = CGet32Bit(BASEIO, VAL_ACK)) != BITSET))
            check = Put32Bit(VAL_ACK, DUAL, BITCLR);
    } /*end if linetype == 'd'*/
} /*end while fgets*/
printf("\nEnd of HSC parameters ASCII file.\n");
check = Put32Bit(END.ASCII, DUAL, BITSET);
fclose(fip);

/***** Read ASCII file with frequency values for SCITEQ or COMSTRON******/
printf("\nReading frequency select values from ASCII file.\n");
if ((fip=fopen("freq.dat", "r")) == NULL) {
    printf("\nError. Cannot open input file for frequency parameters.\n");
    exit(1);
} /*end if fip*/
printf("\nInput file opened for frequency parameters.\n");
while (fgets(fstring, maxlen, fip) != NULL) {
    printf("\nReading frequency data.\n");
    sscanf(fstring, "%c %i %lx", &linetype, &idx, &data);
    if (linetype == 'd')
    {
        switch (idx)
        {
        case 0:
            lo_band = data;
            break;
        case 1:
            up_band = data;
            break;
        case 2:
            mid_band = data;
            break;
        case 3:
            demean_band = data;
            break;
        case 4:
            threeq_band = data;
            break;
        case 5:
            stop_fsel = data;
            break;
        case 6:
            base_f = data;
            break;
        case 7:
            hop_bw = data;
            break;
        }
default:
    printf("\nError in data file index value.");
    exit(1);
} /*end switch (idx)*/
} /*end if linetype*/
} /*end while fgets*/
printf("\nEnd of freq.dat ASCII file.");
fclose(f2p);

/
*****Initialize serial communications*****/

if (ser_select == 1)
{

******Open all communications******
if ((mlocal = open_com()) == BAD_STATION) pabort("\nError in open_com.*");
    printf("\nLocal station is %s", strlwr(mlocal, string));

******Select link to PL synch processor******
if ((ndest = lock_com("SYNC_PROC")) == BAD_STATION)
    pabort("\nBad station lookup");

******Send message to remote terminal that local terminal is ready******
    printf("\nSending 'READY' message to remote terminal.");
    send_com(ndest, STATUS, "Local terminal ready.");
} /*end if (ser_select)*/

} /*end if (ser_select)*/

printf("\nLocal terminal ready.");

Disp_menu();

******Loop operation******

while (stop_PC == BITCLR)
{

****** Step 1: Check for local keyboard input *****

    key_ret = checkkey(ndest); /*Check for local keyboard input*/
    if (key_ret != 0)
    {
        if (key_ret == 1) /*regular exit*/
        {
            stop_PC = BITSET;
            printf("\nProgram halted by local user.");
            if (ser_select == 1)
                send_com(ndest, STATUS, "Program halted at local terminal.");
            break;
        } /*end if key_ret == 1*/
        else if (key_ret == 2) /*exit slow hopping*/
        {
            printf("\nSlow hopping halted.");
            command_given = BITCLR;
            if (ser_select == 1)
                send_com(ndest, STATUS, "Slow hopping halted at local terminal.");
        } /*end else*/
    } /*end if key_ret*/

****** Step 2: Check for remote terminal input (if serial comms enabled)*****

    if (ser_select == 1)
    {
        msg_ret = checkmsg(); /*Check for remote input*/
        switch (msg_ret)
        {
            case 0: /*normal exit*/
                break;
            case 1: /*normal remote exit*/
                stop_PC = BITSET;
                printf("\nProgram halted by remote user.");
                send_com(ndest, STATUS, "Program halted by remote terminal.");
                break;
            case 2: /* slow hopping halted by remote terminal */
                printf("\nSlow hopping halted by remote terminal.");
                send_com(ndest, STATUS, "Slow hopping halted by remote terminal.");
                break;
            case 3: /* Synch response buffer overflow */
                stop_PC = BITSET;
                printf("\nError. Synch resp buffer overflow. Abort.");
                send_com(ndest, STATUS, "Error. Synch resp buffer overflow. Abort.");
                break;
            case 4: /*Error in get_com found*/
                stop_PC = BITSET;
                printf("\nError in serial communications link detected.");
                break;
        } /*end switch (msg_ret)*/
    } /*end if (ser_select)*/

B5
default:
    printf("\nERROR. Invalid return value from check_msg subroutine.\n");
    exit(1);
} /* end switch (msg_ret) */
} /* end if (aer_select) */

****** Step 3: Check for rising edge detect for DL synch ******

flag1 = BITCLR;
if ((flag1 = CTGet32Bit(BASEIO,RIS_DET)) == BITSET)
{
    hop_b4edge = CTGet32Bit(BASEIO,CHK_HOP);
    frm_b4edge = CTGet32Bit(BASEIO,CHK_FRM);
    printf("\nRISING EDGE DETECTED ON FRAME 0 LINE.\n");
    printf("\nValues of hop and frm counters before edge: %li %li, hop_b4edge, frm_b4edge\n");
    check = Put32Bit(RIS_DET, DUAL, BITCLR);
    printf("\nTo disable FR0 detection, enter 'D' \n");
} /* end if (flag1) */

****** Step 4: Check for TMS response re: end of program ******

flag1 = BITCLR;
if ((flag1 = CTGet32Bit(BASEIO,C3DONE)) == BITSET) /* check for TMS response*/
{
    check = Put32Bit(C3DONE, DUAL, BITCLR);
    printf("\nTMS response received and cleared.\n");
    if (operation == local)
        printf("\nCommand executed. Another (y or n)? \n");
    else if (operation == remote)
    {
        command_given = BITCLR;
        if (aer_select == 1)
            send_command(dest, STATUS, "Op complete. Ready for command.\n");
        else if (flag1)
    } /* end if (flag1) */

****** Step 5: Check for transfer of next synch resp to TMS ******

/* if (mode_of_op == 6) /* check for synch mode */
{ /* if ((cmp_res = strcmp(&c_response_arr[resp_rd_ptr][0], "empty") != 0) /* if buffer not empty*/
    if ((flag1 = CTGet32Bit(BASEIO, RDY_4_RESP)) == BITSET)
    {
        sscanf(&c_response_arr[resp_rd_ptr][0], \ "%i %i %i %i %i", &frame, &c0_crsp,
        &c1_crsp, &c0_famps, &c1_famps);
        check = Put32Bit(FRM_REF, DUAL, frame);
        check = Put32Bit(C0_RESP, DUAL, c0_crsp);
        check = Put32Bit(CL_RESP, DUAL, c1_crsp);
        check = Put32Bit(CL_RESP, DUAL, c0_famps);
        check = Put32Bit(CL_RESP, DUAL, c1_famps);
        check = Put32Bit(RESP_AVAL, DUAL, BITSET);
        check = Put32Bit(RDY_4_RESP, DUAL, BITCLR);
        strcopy(&c_response_arr[resp_rd_ptr][0], "empty"); /* clear buffer element*/
        if (resp_rd_ptr == (RESP_BUF_SZ - 1)) /* check for buffer rollover*/
            resp_rd_ptr = 0;
        else
            resp_rd_ptr = resp_rd_ptr + 1;
    } /* end if flag1*/
} /* end if (c_response_arr)*/
/* else if (mode_of_op == 6 */
else

/* if (mode_of_op == 7) /* check for synch mode */
/* if (f_response_arr[resp_rd_ptr] == 999)
{ */
f_est_avg = f_response_arr[resp_rd_ptr]; /* retrieve average fine estimate */
f_response_arr[resp_rd_ptr] = 999; /* clear buffer element */
if (resp_rd_ptr == (RESP_BUF_SZ - 1)) /* update pointer*/
    resp_rd_ptr = 0;
else
    resp_rd_ptr = resp_rd_ptr + 1;
else
    resp_rd_ptr = resp_rd_ptr + 1;
else
    resp_rd_ptr = resp_rd_ptr + 1;
else
    resp_rd_ptr = resp_rd_ptr + 1;
/* else
    check = Put32Bit(FRM_REF, DUAL, f_est_avg);
    check = Put32Bit(PHS_CHANGE, DUAL, phs_change_4_nco);
    check = Put32Bit(STP_AVAL, DUAL, BITSET);
*/
/* }/* end if f_response_arr*/
/* */
/* } /* end else */

B6
***** Step 6: Check for proper transfer of synch resp to TMS, goes with Step 5 *****

/*
 if (mode_of_op == 6)
 {
     if (((flag1 = CTGet32Bit(BASEIO, RESP_ACK)) == BITSET)
         {
             if (check = Put32Bit(RESP_ACK, DUAL, BITCLR);
                 if (check = Put32Bit(REDY_R_RESP, DUAL, BITSET);
                 } /*end if flag1*/
         } /*end if mode_of_op == 6 */
     }
     else
     { /*
         if (mode_of_op == 7)
         {
             if (((flag1 = CTGet32Bit(BASEIO, EST_ACK)) == BITSET)
                 check = Put32Bit(EST_ACK, DUAL, BITCLR);
             } /*end if mode_of_op == 7 */
         } /*end else */
     }

***** Step 7: Check for search range exceeded during CSYNC ******
***** Check for CSYNC_OK condition ******
***** Check for pipeline overflow in synch response buffer ******
***** Check for hyp_log overflow ******
***** Check for FRM_NOT_FOUND when retrieving hypothesis ******
***** Check for FSYN CK condition ******
***** Check for NO_FSYNC condition - non-convergence ******
***** Check for FSTART_FRM for fine sync process ******

flag1 = BITCLR;
if (((flag1 = CTGet32Bit(BASEIO, RNG_XCDED)) == BITSET)
{
    check = Put32Bit(RNG_XCDED, DUAL, BITCLR);
    printf("\nSearch range exceeded encountered.");
    printf("\nDo you want to go back to EL sync? (y/n) ");
    scanf("%s", &u_input);
    switch (u_input)
    {
    case 'y':
    case 'Y':
        int_mode = BITSET;
        mode_of_op = 4;
        send_mode(mode_of_op);
        printf("\nEnabling PR0 detection again.");
        if (ser_select == 1)
            send_command(dest, STATUS, "LOC - back to PR0 detect");
        command_given = BITSET;
        break;
    case 'n':
    case 'N':
        stop_PC = BITSET;
        printf("\nExiting program now");
        break;
    default:
        printf("\nInvalid response. Aborting program.");
        stop_PC = BITSET;
        break;
    } /*end switch u_input*/
} /*end if ...RNG_XCDED*/
else /****/
{ /*
    flag1 = BITCLR;
    if (((flag1 = CTGet32Bit(BASEIO, CSYNC_OK)) == BITSET)
    {
        check = Put32Bit(CSYNC_OK, DUAL, BITCLR);
        printf("\nCoarse synch achieved.");
        printf("\nDo you want to go to fine sync? (y/n) ");
        scanf("%s", &u_input);
        switch (u_input)
        {
        case 'y':
        case 'Y':
            int_mode = BITSET;
            mode_of_op = 7;
            send_mode(mode_of_op);
            printf("\nProceeding to fine sync.");
            if (ser_select == 1)
                send_command(dest, STATUS, "LOC - going to fine sync");
            command_given = BITSET;
            break;
        case 'n':
        case 'N':
            stop_PC = BITSET;
            printf("\nExiting program now");
            break;
        default:
        */
*/
*/
B7
printf("Invalid response. Aborting program.");
stop_PC = BITSET;
break;
} //end switch u_input/
} /*end if ...CSYNC_OK*/
else /*skip*/
{
flag1 = BITCLR;
if (get32Bit(BASEIO, PLLINE_FLAG)) == BITSET
{
    check = Put32Bit(PLINE_FLAG, DUAL, BITCLR);
    printf("Synchron response buffer pipeline overflow in TMS. Exiting program.");
    stop_PC = BITSET;
} /*end if flag1 ... PLINE_FLAG*/
else /*skip*/
flagl = BITCLR;
if (get32Bit(BASEIO, TOO_MANY_HYPS)) == BITSET
{
    check = Put32Bit(TOO_MANY_HYPS, DUAL, BITCLR);
    printf("Too many hypotheses buffer overflow. Exiting program.");
    stop_PC = BITSET;
} /*end if flag1 ... TOO_MANY_HYPS*/
else /*skip*/
flagl = BITCLR;
if (get32Bit(BASEIO, FRM_NOT_FOUND)) == BITSET
{
    check = Put32Bit(FRM_NOT_FOUND, DUAL, BITCLR);
    printf("Unable to retrieve hypothesis to verify coarse sync. Aborting.");
    stop_PC = BITSET;
} /*end if flag1 ... FRM_NOT_FOUND*/
else /*skip*/
flagl = BITCLR;
if (get32Bit(BASEIO, PSTART_AVAIL)) == BITSET
{
    frm_ref4_file = get32Bit(BASEIO, PSTART_FRM);
    check = Put32Bit(PSTART_AVAIL, DUAL, BITCLR);
    check = Put32Bit(PSTART_ACK, DUAL, BITCLR);
} /*end if flag1 ... PSTART_AVAIL*/
else /*skip*/
flagl = BITCLR;
if (get32Bit(BASEIO, PSYNC_OK)) == BITSET
{
    check = Put32Bit(PSYNC_OK, DUAL, BITCLR);
    printf("Fine synch achieved. Ready to transmit data. \n\n");
    stop_PC = BITSET; /* TEST ONLY */
} /*end if flag1 ... PSYNC_OK*/
else /*skip*/
flagl = BITCLR;
if (get32Bit(BASEIO, NO_FSYNC)) == BITSET
{
    check = Put32Bit(NO_FSYNC, DUAL, BITCLR);
    printf("Fine estimates not converging.");
    printf("Do you want to go back to coarse sync? (y/n) ");
    scanf("%c", &u_input);
    switch (u_input)
    {
        case 'y':
            mode_of_op = 6;
            send_mode(mode_of_op);
            hypothesis = 0;
            check = Put32Bit(USER_HYP, DUAL, hypothesis);
            check = Put32Bit(HYP_avail, DUAL, BITSET);
            flag = BITCLR;
            while ((flagl = get32Bit(BASEIO, HYP_ACK)) == BITCLR)
                printf("Waiting hyp acknowledgement");
            check = Put32Bit(HYP_ACK, DUAL, BITCLR);
            if (ser_select == 1)
                send_command, STATUS, "LOC - going back to csync");
            break;
        case 'n':
            case 'n':
                printf("Exiting program");
                stop_PC = BITSET;
                break;
        default:
            printf("Invalid response. Exiting program now.");
            stop_PC = BITSET;
            break;
    } /*end switch*/
} /*end if flag1 ... NO_FSYNC*/
/*end else */
B8
****** Step 8: Check for data underflow in GT processor i/f board ******

flagl = BITCLR;
if ((flagl = CTGet32Bit(BASEIO, UFLO_CDYN)) == BITSET)
    printf("Data underflow condition received. Aborting program.");
    stop_PC = BITSET;
} //end if flag UFLO_CDYN */

} //end while (stop_PC) */

****** Disable TMS interrupts before exiting ******

/*check = Put32Bit(STOP_INT, DUAL, BITSET);
/*while (((flagl=CTGet32Bit(BASEIO,STOP_INT))==BITSET)); */

if (ser_select == 1)
    { while (ready_com[ndest]==0) /*wait for transmit buffer to be ready*/
        checkmsg();
        send_com[ndest, STATUS, "End of GT program."]; 
        while (chk_time[0] != 0);
        close_com();
    } //end if (ser_select)*/

printf("\nEnd of C program.");

} //end of main/*

******************************************************************************
unsigned long CTGet32Bit(int baseadr, long loc)
******************************************************************************
{

    unsigned int low;
    unsigned int high;
    int C30port, commreg, hicomreg;
    unsigned long total;

    C30port = baseadr;
    commreg = C30port + 0;
    hicomreg = C30port + 2;
    SetAddr(loc);
    CntRDis();
    low = inpw(commreg);
    high = inpw(hicomreg);
    total = ((long)high<<16) + low;
    return(total);
}

} //end CTGet32Bit()*/

**************
void Disp_menu()
**************
{
    printf("\n****************************");
    printf("\n CW test - 7-11 April 1997 **");
    printf("\n GT Synch Processor Menu **");
    printf("\n Enter one of the following: **");
    printf("\n **");
    printf("\n [L] : Go to lower edge of hop BW **");
    printf("\n [U] : Go to upper edge of hop BW **");
    printf("\n [M] : Go to middle of hop BW **");
    printf("\n [Q] : Go to one quarter mark of hop BW **");
    printf("\n [T] : Go to three quarter mark of hop BW **");
    printf("\n [F] : Go to specific frequency **");
    printf("\n [R] : Go to RUN mode **");
    printf("\n [S] : Slowly cycle through hop BW **");
    printf("\n [E] : Enable interrupt/FRO detection **");
    printf("\n [D] : Disable interrupt/FRO detection **");
    printf("\n [C] : Coarse synchronization test **");
    printf("\n [W] : Fine synchronization test **");
    printf("\n [X] : Exit program or stop slow hopping (option 'S') **");
    if (int_mode == BITCLR)
        printf("\n THS interrupts/FRO detection is currently disabled. **");
    else
        printf("\n THS interrupts/FRO detection is currently enabled. **");

B9
printf("Enter selection: ");
return;
} /*end Disp_menu()*/

int checkkey(int dest)
{ /*character from keyboard*/
  unsigned long user_f, freq_val;
  if (kbhit() != 0) /*is a key pressed*/
  { 
    operation = local;
    c = getch();
    switch (command_given)
    {
      case 0:
        switch (c)
        {
          case 'L':
            case 'l':
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = lo_band;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - go low command sent.");
              command_given = BITSET;
              break;
            case 'U':
              case 'u':
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = up_band;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - go high command sent.");
            command_given = BITSET;
            break;
          case 'M':
            case 'm':
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = mid_band;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - go mid command sent.");
              command_given = BITSET;
              break;
          case 'Q':
            case 'q':
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = oneq_band;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - go 1/4 command sent.");
            command_given = BITSET;
            break;
            case 'T':
            case 't':
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = threeq_band;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - go 3/4 command sent.");
            command_given = BITSET;
            break;
          case 'F':
            case 'f':
              printf("Enter frequency in 100Hz: ");
              scanf("%li", &user_f);
              mode_of_op = 1;
              send_mode(mode_of_op);
              freq_val = user_f;
              send_f(freq_val);
              if (ser_select == 1)
                send_com(ndest, STATUS, "LOC - user freq command sent.");
            command_given = BITSET;
            break;
        case 'R':
          case 'r':
            mode_of_op = 2;
            send_mode(mode_of_op);
            printf("Going to RUN mode.");

B10
if (ser_select == 1)
    send_com(ndest, STATUS, "LOC - RUN command sent.");
command_given = BITSET;
broke;  
case 'S':
case 's':
    mode_of_op = 3;
    printf("\nEntry changes mode of_op to : %i", mode_of_op);
    send_mode(mode_of_op);
    printf("\nPlease enter dwell time in seconds: ");
    scanf("%i", &dwell_time);
    printf("\nPlease enter hop increment 0-16777215: ");
    scanf("%i", &hop_incr);
    dwell_cycles = (unsigned long)((dwell_time/60.0e-9) + 0.5);
    check = Put32Bit(DWELL_VAL, Dual, dwell_cycles);
    check = Put32Bit(DWELL_AVAIL, Dual, BITSET);
    flagl = BITCLR;
    while ((flagl = CTGet32Bit(BASEIO, DWELL_ACK)) != BITSET) {
        printf("Waiting for TMS to ack receipt of dwell. ");
        printf("\nDwell transferred.\n");
        check = Put32Bit(DWELL_ACK, Dual, BITCLR);
        check = Put32Bit(INCR_VAL, Dual, hop_incr);
        check = Put32Bit(INCR_AVAIL, Dual, BITSET);
        flagl = BITCLR;
        while ((flagl = CTGet32Bit(BASEIO, INCR_ACK)) != BITSET) {
            printf("\nHop increment transferred.\n");
            check = Put32Bit(INCR_ACK, Dual, BITCLR);
            printf("\nTo stop slow hopping through BN, type 'X' ");
        }
        if (ser_select == 1)
            send_com(ndest, STATUS, "LOC - sweep command sent.");
        command_given = BITSET;
        broke;
    }
    break;
    }
    break;
    case 'D':
    case 'd':
        int_mode = BITCLR;
        mode_of_op = 5;
        send_mode(mode_of_op);
        printf("\nDisabling FR0 detection.\n");
        if (ser_select == 1)
            send_com(ndest, STATUS, "LOC - disabling FR0 detection.");
        command_given = BITSET;
    broke;
    case 'E':
    case 'e':
        int_mode = BITSET;
        mode_of_op = 4;
        send_mode(mode_of_op);
        printf("\nEnabling FR0 detection.\n");
        if (ser_select == 1)
            send_com(ndest, STATUS, "LOC - enabling FR0 detection.");
    command_given = BITSET;
    broke;
    case 'C';
    case 'c';
    mode_of_op = 6;
    send_mode(mode_of_op);
    printf("\nSync probes are transmitted starting at hop 288 by default.\n");
    printf("\nPlease enter desired hypothesis offset or 0 if none: ");
    scanf("%i", &hypothesis);
    printf("\nHyps will be at 0,1,-1,2,-2,...from hop 288 \(\text{if no offset}\)\n");
    check = Put32Bit(USEX_HYP, Dual, hypothesis);
    check = Put32Bit(HYP_AVAIL, Dual, BITSET);
    printf("\nHYP_AVAIL signal sent to TMS.\n");
    flagl = BITCLR;
    while ((flagl = CTGet32Bit(BASEIO, HYP_ACK)) == BITCLR) {
        printf("\nWaiting for hypothesis acknowledge. ");
        printf("\nHypothesis transferred.\n");
        check = Put32Bit(HYP_ACK, Dual, BITCLR);
    if (ser_select == 1)
        send_com(ndest, STATUS, "LOC - coarse synch initiated.");
    command_given = BITSET;
    broke;
    case 'W';
    case 'w';
    mode_of_op = 7;
    send_mode(mode_of_op);
    printf("\nAttempting fine synch. \n\n");
    if (ser_select == 1)
        send_com(ndest, STATUS, "LOC - fine synch initiated");
    command_given = BITSET;
    broke;
    case 'X';
    case 'x';
    if (mode_of_op == 3) {
        check = Put32Bit(STOP_MODE, Dual, BITSET);
        flagl = BITSET;
    B11
while ((flag1 = CTGet32Bit(BASEIO, STOP_MOD3)) != BITCLR);
printf("nNot stopped TMS in mode3");
mode_of_op = 0;
return 2;
}
else
return 1;
}
default:
printf("nInvalid entry. Try again.");
Disp_menu();
break;
} /*end switch (c), case 0 for command given*/
break;
case 0:
switch(c)
{
  case 'Y':
  case 'y':
    Disp_menu();
    command_given = BITCLR;
    if (ser_select == 1)
      send_com(ndest, STATUS, "LOC - Ready for next command.");
    break;
  case 'N':
  case 'n':
    if (ser_select == 1)
      send_com(ndest, STATUS, "LOC - user's stopping program.");
    return 1;
  case 'X':
  case 'x':
    if (mode_of_op == 3)
    {
      check = Put32Bit(STOP_MOD3, DUAL, BITSET);
      flag1 = BITSET;
      while ((flag1 = CTGet32Bit(BASEIO, STOP_MOD3)) == BITCLR);
      printf("nNot stopped TMS in mode3");
      mode_of_op = 0;
      if (ser_select == 1)
        send_com(ndest, STATUS, "LOC - exit slow hopping command.");
      return 2;
    }
    else
      return 1;
    default:
    printf("nInvalid entry.");
    printf("nPlease enter, y, n, or x to exit immediately");
    break;
} /*end switch (c), case 0 for command given*/
break;
} /*end switch (command_given)*/
} /*end if (khhbit)*/
} /*end checkkey(dest)*/

************
int checkmsg()

************
{
  int mstat = get_com(&mtype, &mfrom, mdata);
  if (mstat == VALID_MSG)
    {
      print("Message received from %s", mfrom.string);
      print("%s\n\n", mtype.string, mdata);
      proc_ret = proc_msg(mtype, mdata);
      return(proc_ret);
    }
  } /*end if mstat*/
else if (mstat == COMM_ERR)
  {
    print("Comm error with %s: %s", mfrom.string, mdata);
  } /*end else if mstat==COMM_ERR*/
else if (mtype == QUIT)
  {
    if (mtype == TOTAL)
      {
        printf("Too many communication errors.");
        return 4;
      }
  } /*end if mtype==QUIT*/
else if (mtype == CONSEC)
  {
    printf("Too many consecutive communication errors with %s", mfrom.string);
  }
return 4;
}
else if (mtype == BREAK)
{
    printf("\nBreak detected.");
} /*end if mtype==BREAK*/
return 1;
} /*end if mstat==QUIT*/
return 0;
} /*end checkmsg()*/

/***************************/
void pabort(char *msg)
/***************************/
{
    printf("\n%s", msg);
    close_com();
    exit(2);
} /*end pabort()*/

/***************************************************/
void send_mode(int mode_sel)
/***************************************************/
{
    check = Put32Bit(MODE, DUAL, (long)mode_sel); /*Transfer mode*/
    check = Put32Bit(MODE_AVAI, DUAL, BITSET); /*Signal TMS that mode is avail*/
    flagl = BITCLR;
    while (((flagl = CTGet32Bit(BASEIO, MODE_ACK)) != BITSET){
        if (operation == local) {
            printf("\nWaiting for TMS to ack mode transfer, flag1: \n", flagl);
        } /*end while flag1*/
        if (operation == local) {
            printf("\nMode transfer acknowledged.\n");
        } /*end if operation*/
        check = Put32Bit(MODE_ACK, DUAL, BITCLR);
        return;
    } /*end send_mode(mode_sel)*/

/***************************************************/
void send_f(uns long f_val)
/***************************************************/
{
    /********Compute appropriate value for LD_HOP in HSC*********/
    hop_par = ( ((double)(f_val - base_f) / (double)hop_bw) * (double)MAX_HOP_NUM ) + 0.5;
    check = Put32Bit(F_AVAIL, DUAL, BITSET); /*signal TMS that frequency is avail*/
    flagl = BITCLR;
    while (((flagl = CTGet32Bit(BASEIO, F_ACK)) != BITSET){
        if (operation == local) {
            printf("\nWaiting for TMS to ack freq transfer, flag1: \n", flagl);
        } /*end while flag1*/
        if (operation == local) {
            printf("\nFrequency select transfer acknowledged.\n");
        } /*end if operation*/
        check = Put32Bit(F_ACK, DUAL, BITCLR);
        return;
    } /*end send_f(f_val)*/

/***************************************************/
int proc_msg(int type, char mcontent[220])
/***************************************************/
{
    char cmd[20], param1[20], param2[20];

    operation = remote;
    if (type == COMMAND)
    {
        /***** Break up message string into separate fields *****
        sscanf(mcontent, "%s %s %s", cmd, param1, param2);
        if (strcmp(cmd, "Set_freq") == 0)
        {
            mode_of_op = 1;
            send_mode(mode_of_op);
            freq_val = (long int)Rd_value(param1);
            send_f(freq_val);
            printf("\nRemote user frequency sent to TMS.\n");
            command_given = BITSET;
        } /*end if strcmp = Sec_freq*/
        else if (strcmp(cmd, "Go_to_run") == 0)
        {
mode_of_op = 2;
send_mode(mode_of_op);
printf("\nRemote RUN command sent to TMS.\n");
command_given = BITSET;
} /* end if strncmp = go_to_run*/
else if (strncmp(cmd, "Dwell_hop", 9)==0)
{
    mode_of_op = 3;
send_mode(mode_of_op);
dwell_time = Rd_value[param1];
dwell_cycles = (unsigned long)((dwell_time/60.0e-9) + 0.5);
check = Put32Bit(DWELL_VAL, DUAL, dwell_cycles);
check = Put32Bit(DWELL_AVAIL, DUAL, BITSET);
flag1 = BITCLR;
while (((flag1 = CTGet32Bit(BASEIO, DELL_ACK)) != BITSET) /*wait*/;
    check = Put32Bit(DWELL_ACK, DUAL, BITCLR);
hop_incr = (long int)Rd_value[param2];
check = Put32Bit(INCR_VAL, DUAL, hop_incr);
check = Put32Bit(INCR_AVAIL, DUAL, BITSET);
flag1 = BITCLR;
while (((flag1 = CTGet32Bit(BASEIO, INCR_ACK)) != BITSET) /*wait*/;
    check = Put32Bit(INCR_ACK, DUAL, BITCLR);
printf("\nRemote slow hop command sent to TMS.\n");
command_given = BITSET;
} /* end if strncmp = dwell_hop*/
else if (strncmp(cmd, "Exit", 4)==0)
{
    if (mode_of_op == 3)
    {
        check = Put32Bit(STOP_MOD3, DUAL, BITSET);
        flag1 = BITSET;
        while (((flag1 = CTGet32Bit(BASEIO, STOP_MOD3)) != BITCLR);
            printf("\nRemote stop TMS in mode3\n");
            mode_of_op = 0;
            send_com(dest, STATUS, "Slow hop terminated.\n");
            command_given = BITCLR;
            return 2;
        }
    }
    else
        return 1;
} /* end if strncmp = exit*/
else if (strncmp(cmd, "Enable_FR0",10)==0)
{
    mode_of_op = 4;
send_mode(mode_of_op);
printf("\nRemote enable FR0 detect sent to TMS.\n");
command_given = BITSET;
} /* end if strncmp = Enable_FR0*/
else if (strncmp(cmd, "Disable_FR0",11)==0)
{
    mode_of_op = 5;
send_mode(mode_of_op);
printf("\nRemote disable FR0 detect sent to TMS.\n");
command_given = BITSET;
} /* end if strncmp = Disable_FR0*/
else if (strncmp(cmd, "Go_2_csynic",10)==0)
{
    mode_of_op = 6;
send_mode(mode_of_op);
}

/*****NEED TO ADD IN INPUT FOR HYPOTHESIS OFFSET, ETC...*****/
prompt("\nRemote CSYNIC command sent to TMS.\n");
command_given = BITSET;
} /* end if strncmp = Go_2_csynic*/
else
printf("\nWrong command statement: %s\n", mcontent);
send_com(dest, STATUS, "Unknown command.\n");
} /* end if type = COMMAND*/
else if (type == STATUS)
{
if (isdigit(mcontent[0]) == 0) /* find out if lst is a digit */
{
    printf("\nStatus message processed: %s\n", mcontent); /*print status message*/
    sscanf(mcontent, "%li", &frame);
    if (mode_of_op == 6) /* check for CSYNIC mode */
    {
        if (((frame<4) == 0)) /* check for frame a mult of 4 */
        {
            if (strcmp(ret = strncmp(kc_resn_arr[resp_wr_ptr][0],"empty")) == 0)
            {
                strcpy(kc_resn_arr[resp_wr_ptr][0],mcontent); /*store synch resp*/
                sscanf(mcontent, "%li %li %li %li", &frame,&kc_crap,kc_crsp,
                &kc_frap,kc_fcrsp);
                printf("\nFrame number, coarse resp, fine resp: %li %li %li %li,\n                frame,kc_crap,kc_crsp,kc_frap,kc_fcrsp);
                if (resp_wr_ptr == (RESP_BUF_SIZE -1)) /*update pointer*/
            }
        }
    }

B14
resp_wr_ptr = 0;
else
resp_wr_ptr = resp_wr_ptr + 1;
printf("\nSynch response stored.\n");
} /*end if sync_ptr*/
else
{
printf("\nERROR. Overflow of synch response buffer in PC.\n");
return 3;
} /*end else*/
} /*end if frame*/
} /*end if mode_of_op == 6*/
else
{
if (mode_of_op == 7)
{
if (((frame&4) == 0) && (frame >= frm_ref_4_fine))
{
sscanf(mccontent, "%44%44%44%44%44%44", &lframe, &c0_crsp, &c1_crsp, &c0_frsp, &c1_frsp);
cum_fineResp = cum_fineResp + c1_frsp; /*cumul estimates for user 1*/
fine_cnt = fine_cnt + 1;
if (fine_cnt == 10)
{
if (cum_fineResp < 0)
filt_avg = ((float)cum_fineResp/10.0) + 0.5; /* calc avg fine est */
else
filt_avg = ((float)cum_fineResp/10.0) - 0.5;
f_est_avg = (long int)filt_avg;
cum_fineResp = 0; /* reset fine resp accumulator */
fine_cnt = 0; /* reset fine estimate received count */
if (f_resp_arr[resp_wr_ptr] == 999)
{
if (resp_wr_ptr == (RESP_BUF_SIZE - 1)) /* update pointer */
resp_wr_ptr = 0; else
resp_wr_ptr = resp_wr_ptr + 1;
} /*end if resp_arr */
else
printf("\nERROR. Overflow of synch response buffer.\n");
return 3;
} /*end else*/

adjustment = (float)f_est_avg/(62.5 * 320.0); /*COMPUTE NCO CHANGE NECESSARY*/
phs_change_4_nco = (long int)((1.0+adjustment)*NCO_VAL);
if (f_resp_arr[resp_wr_ptr] == 999)
{
if (resp_wr_ptr == (RESP_BUF_SIZE - 1)) /* update pointer */
resp_wr_ptr = 0; else
resp_wr_ptr = resp_wr_ptr + 1;
} /*end if resp_arr */
else
printf("\nERROR. Overflow in synch resp buffer.\n");
return 3;
} /*end else*/
} /*end if frame...*/
} /*end if mode_of_op == 7*/
} /*end else*/
} /*end if isdigit...*/
else
printf("instatus message: %s", mccontent); /*print status message*/
} /*end else if type*/
return 0;
} /* end proc_msg(mccontent)*/

/******************************/
float Rd_value(char par[20])
******************************/
{
float number;

/* Find out whether it's a digit */
if (isdigit(par[0]) == 0) { /* first char is not a digit */
if (strcmp(par, "low", 3)==0)
return((float)10_band);
else if (strcmp(par, "high", 4)==0)
return((float)bg_band);
else if (strcmp(par, "mid", 3)==0)
return((float)mid_band);
else if (strcmp(par, "oneq", 4)==0)
return((float)one_band);
else
printf("instatus message: %s", par); /*print status message*/
} /*end else if type*/
return 0;
} /* end proc_msg(mccontent)*/

/******************************/
float Rd_value(char par[20])
******************************/
{
float number;

/* Find out whether it's a digit */
if (isdigit(par[0]) == 0) { /* first char is not a digit */
if (strcmp(par, "low", 3)==0)
return((float)10_band);
else if (strcmp(par, "high", 4)==0)
return((float)bg_band);
else if (strcmp(par, "mid", 3)==0)
return((float)mid_band);
else if (strcmp(par, "oneq", 4)==0)
return((float)one_band);
else
printf("instatus message: %s", par); /*print status message*/
} /*end else if type*/
return 0;
} /* end proc_msg(mccontent)*/
return((float)oneq_band);
else if(strcmp(par, "threeq") && !0)
    return((float)threeq_band);
else {
    return(0.0);
} /*end else*/
} /*end if isdigit*/
else {
    scanf(par, "%f", &number);
    return(number);
} /*end else*/
}/*end Rd_value subroutine*/

**************
void Init_DMEM()  
**************
{
    check = Put32Bit(C3DONE, DUAL, BITCLR);
    check = Put32Bit(MODE_AVAIL, DUAL, BITCLR);
    check = Put32Bit(MODE, DUAL, BITCLR);
    check = Put32Bit(MODE_ACK, DUAL, BITCLR);
    check = Put32Bit(F_AVAIL, DUAL, BITCLR);
    check = Put32Bit(F_VALUE, DUAL, BITCLR);
    check = Put32Bit(F_ACK, DUAL, BITCLR);
    check = Put32Bit(PHS_AVAIL, DUAL, BITCLR);
    check = Put32Bit(PHS_RNDED, DUAL, BITCLR);
    check = Put32Bit(PHS_ACK, DUAL, BITCLR);
    check = Put32Bit(DWELL_AVAIL, DUAL, BITCLR);
    check = Put32Bit(DWELL_Val, DUAL, BITCLR);
    check = Put32Bit(DWELL_ACK, DUAL, BITCLR);
    check = Put32Bit(VAL_RDY, DUAL, BITCLR);
    check = Put32Bit(VAL_ID, DUAL, BITCLR);
    check = Put32Bit(VALUE, DUAL, BITCLR);
    check = Put32Bit(VALUE_ACK, DUAL, BITCLR);
    check = Put32Bit(END_ASCII, DUAL, BITCLR);
    check = Put32Bit(STOP_MODE, DUAL, BITCLR);
    check = Put32Bit(INCR_AVAIL, DUAL, BITCLR);
    check = Put32Bit(INCR_VALUE, DUAL, BITCLR);
    check = Put32Bit(INCR_ACK, DUAL, BITCLR);
    check = Put32Bit(STOP_INT, DUAL, BITCLR);
    check = Put32Bit(RS_DET, DUAL, BITCLR);
    check = Put32Bit(HYP_AVAIL, DUAL, BITCLR);
    check = Put32Bit(HYP_VALUE, DUAL, BITCLR);
    check = Put32Bit(HYP_ACK, DUAL, BITCLR);
    check = Put32Bit(DBUG_AVAIL, DUAL, BITCLR);
    check = Put32Bit(DBUG_VALUE, DUAL, BITCLR);
    check = Put32Bit(DBUG_ACK, DUAL, BITCLR);
    check = Put32Bit(CHK_AVAIL, DUAL, BITCLR);
    check = Put32Bit(CHK_VALUE, DUAL, BITCLR);
    check = Put32Bit(CHK_ACK, DUAL, BITCLR);
    check = Put32Bit(FRM_RESP, DUAL, BITCLR);
    check = Put32Bit(CX_RESP, DUAL, BITCLR);
    check = Put32Bit(CX_RESP, DUAL, BITCLR);
    check = Put32Bit(DL_RESP, DUAL, BITCLR);
    check = Put32Bit(RESERVED, DUAL, BITCLR);
    check = Put32Bit(RESERVED, DUAL, BITCLR);
    check = Put32Bit(RDY_4_RESP, DUAL, BITSET);
    check = Put32Bit(RNG_XCED, DUAL, BITCLR);
    check = Put32Bit(CSYNC, DUAL, BITCLR);
    check = Put32Bit(CHK_FMB, DUAL, BITCLR);
    check = Put32Bit(LOG_END, DUAL, BITCLR);
    check = Put32Bit(PLINE_FLAG, DUAL, BITCLR);
    check = Put32Bit(DATE, DUAL, BITCLR);
    check = Put32Bit(INDEX, DUAL, BITCLR);
    check = Put32Bit(DAT_VALUE, DUAL, BITCLR);
    check = Put32Bit(DAT_ACK, DUAL, BITCLR);
    check = Put32Bit(END_FILE, DUAL, BITCLR);
    check = Put32Bit(TOO_MANY_HYPS, DUAL, BITCLR);
    check = Put32Bit(FRM_NOT_FOUND, DUAL, BITCLR);
    check = Put32Bit(FSTART_AVAIL, DUAL, BITCLR);
    check = Put32Bit(FSTART_FRM, DUAL, BITCLR);
    check = Put32Bit(FSTART_ACK, DUAL, BITCLR);
    check = Put32Bit(EST_AVAIL, DUAL, BITCLR);
    check = Put32Bit(FINE_EST, DUAL, BITCLR);
    check = Put32Bit(FINS_CHANGE, DUAL, BITCLR);
    check = Put32Bit(EST_ACK, DUAL, BITCLR);
    check = Put32Bit(NO_FSYNC, DUAL, BITCLR);
    check = Put32Bit(FSYNC_OK, DUAL, BITCLR);
    check = Put32Bit(UPLCDUTDOWN, DUAL, BITCLR);
} /*end Init_DMEM subroutine*/
B2. DSP main program

******************************************************
* Program Name: TMS_UL2.ASM
* Author: C. Tom
* Date edited: 26 February 1998
* Description: This program performs uplink GT processor functions including
* transmission of CW tones, slow sweeping across entire BW,
* detecting FR0 pulse for DL sync confirmation, transmitting
* of coarse sync probes and achieving coarse synchronization.
* This version has the parameters for the GT processor
* downloaded from an ASCII data file.
******************************************************

**** Subroutine declarations ****
.globl             CNG_COM_DISP
.globl             CHOOSE_F
.globl             COMMAND_CLK
.globl             CR8E_SYNC
.globl             Disableint
.globl             DSPFILYLIP
.globl             Enableint
.globl             FINE_SYNC
.globl             GT_ISR
.globl             HSC_INIT
.globl             ld_param_HSC
.globl             NRDY_low_loop
.globl             Rd_GTparam
.globl             Rd_HSCParam
.globl             SLOW_HOP
.globl             START_NCO

**** Miscellaneous constants ****
XFO_EN            .set  2h
XP_SET            .set  6h
XP_CLR            .set  0fffbh
BITCLR            .set  0
BITSET            .set  1
MODE1             .set  1 ;mode 1: CW frequency transmit
MODE2             .set  2 ;mode 2: RUN mode
MODE3             .set  3 ;mode 3: sweep alloc bw with dwell
MODE4             .set  4 ;mode 4: FR0 enable mode
MODE5             .set  5 ;mode 5: FR0 disable mode
MODE6             .set  6 ;mode 6: CSYNC mode
MODE7             .set  7 ;mode 7: FSYNC mode
P_STOP             .set  1234h
MASK16            .set  0fffbh

.data              gt_vars

* Indices for gt_vars parameters array
NUM_HOP            .set  0
MAX_FRM            .set  1
MAX_HOP            .set  2
PR0 START          .set  3
SRC_H Lim          .set  4
TIMES_4_CONFH      .set  5
RESP_BUF_SIZ       .set  6
MIN_DET_2_VER      .set  7
NUM_RETRANSMTS     .set  8
LIM_10             .set  9
MIN_4_CONV         .set  10
MAX_ATTEMPTS       .set  11

.data              HSC_array
HSC_addr           .word  HSC_array
BIT80_H_1          .word  3b
BIT22_11           .word  00000000

.data              BEGINSTACK
BEGINSTACK         .word  809800h ;Address of beginning of stack
Pribus             .set  808864h ;Address of primary bus control register
Secbus             .set  808000h ;Address of secondary bus control register

Pribusval          .set  800h
Secbusval          .set  0
PR0_reg_init       .set  1800h ;bit to clear/enable CACHE, disable OVM
* N.B. After RESET, following registers are initialized to zero
  * ST - CPU status register
  * IE - CPU/DMA interrupt enable flags
  * IF - CPU interrupt flags
  * IOF - I/O flags

**** Data page pointers ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTPG</td>
<td>0</td>
</tr>
<tr>
<td>DPMEMPG</td>
<td>3h</td>
</tr>
<tr>
<td>BSSPG</td>
<td>80h</td>
</tr>
<tr>
<td>DATAPG</td>
<td>0h</td>
</tr>
<tr>
<td>DPLANKPG</td>
<td>80h</td>
</tr>
<tr>
<td>BUSPG</td>
<td>80h</td>
</tr>
</tbody>
</table>

**** DPMEM addresses ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPBASE</td>
<td>30000h</td>
</tr>
<tr>
<td>C3DOONE</td>
<td></td>
</tr>
<tr>
<td>MODE_AVAIL</td>
<td></td>
</tr>
<tr>
<td>MODE</td>
<td></td>
</tr>
<tr>
<td>MODE_ACK</td>
<td></td>
</tr>
<tr>
<td>F_AVAL</td>
<td></td>
</tr>
<tr>
<td>F_VALUE</td>
<td></td>
</tr>
<tr>
<td>F_ACK</td>
<td></td>
</tr>
<tr>
<td>PHS_AVAL</td>
<td></td>
</tr>
<tr>
<td>PHS_RDND</td>
<td></td>
</tr>
<tr>
<td>PHS_ACK</td>
<td></td>
</tr>
<tr>
<td>DWWEL_VAL</td>
<td></td>
</tr>
<tr>
<td>DWWEL_ACK</td>
<td></td>
</tr>
<tr>
<td>VAL_ADD</td>
<td></td>
</tr>
<tr>
<td>VAL_IDX</td>
<td></td>
</tr>
<tr>
<td>VALUE</td>
<td></td>
</tr>
<tr>
<td>VAL_ACK</td>
<td></td>
</tr>
<tr>
<td>END_ASCII</td>
<td></td>
</tr>
<tr>
<td>STOP_CMD3</td>
<td></td>
</tr>
<tr>
<td>INCR_AVAL</td>
<td></td>
</tr>
<tr>
<td>INCR_VAL</td>
<td></td>
</tr>
<tr>
<td>INCR_ACK</td>
<td></td>
</tr>
<tr>
<td>STOP_INT</td>
<td></td>
</tr>
<tr>
<td>RIS_DET</td>
<td></td>
</tr>
</tbody>
</table>

* DPBASE+15h to DPBASE+28h, AND DPBASE+29h to DPBASE + 2Fh used in CSYNC.ASM

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHK_HOP</td>
<td>DPBASE + 2Ch</td>
</tr>
<tr>
<td>CHK_FRM</td>
<td>DPBASE + 2Dh</td>
</tr>
<tr>
<td>DAT_AVAIL</td>
<td>DPBASE + 30h</td>
</tr>
<tr>
<td>INDEX</td>
<td>DPBASE + 31h</td>
</tr>
<tr>
<td>DAT_VALUE</td>
<td>DPBASE + 32h</td>
</tr>
<tr>
<td>DAT_ACK</td>
<td>DPBASE + 33h</td>
</tr>
<tr>
<td>END_FILE</td>
<td>DPBASE + 34h</td>
</tr>
</tbody>
</table>

**** BER i/f board addresses ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD_BER</td>
<td>800009h</td>
</tr>
<tr>
<td>STAT_BER</td>
<td>800009h</td>
</tr>
<tr>
<td>BER_DAT_port</td>
<td>800008h</td>
</tr>
</tbody>
</table>

**** BER i/f board commands ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BER_SWres</td>
<td>8000h</td>
</tr>
<tr>
<td>BER_Dfault</td>
<td>0h</td>
</tr>
</tbody>
</table>

**** GT i/f board addresses ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND</td>
<td>800004h</td>
</tr>
<tr>
<td>STATUS</td>
<td>800004h</td>
</tr>
<tr>
<td>NCO_CMD</td>
<td>800005h</td>
</tr>
<tr>
<td>INTRPT_PORT</td>
<td>800005h</td>
</tr>
<tr>
<td>HSC_PORT</td>
<td>800004h</td>
</tr>
<tr>
<td>FSK_FRM</td>
<td>800007h</td>
</tr>
</tbody>
</table>

**** GT i/f board commands ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW_RES_G</td>
<td>1</td>
</tr>
<tr>
<td>Default_CMD</td>
<td>5</td>
</tr>
</tbody>
</table>

**** HSC commands and miscellaneous ****

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
</tr>
</tbody>
</table>
HSC_endhop .word 0xffffh ; 16777215 (max hop number for HSC)
SYNC_BIT .set 2 ; SYNC on bit D1 of GT status
NRDY_BIT .set 4 ; bit D2 on GT status (DSPLINK)
STOP_HSC .set 0
RUN_HSC .set 8000h
CHG_TMEMD .set 0501h
CHG_HOP .set 0500h
LD_LATCH .set 0300h
LD_BASE .set 030Ch
LD_BSCALE .set 0306h
LD_DCFP .set 0310h
LD_FCSPACE .set 030Ah
LD_FLAGS .set 031Ah
LD_FCRGB .set 0308h
LD_HOP .set 0304h
LD_LOSTI .set 0312h
LD_LOCOM .set 030Eh
LD_OFFSET .set 0314h
LD_TIMEL .set 0316h
ULSYNC_CMD .set 500Bh ; GO TO ULSYNC MODE, channel 1, bin 3
ULGO_BASE .set 200h ; BASE VALUE OF ULGO COMMAND FOR CSYNC

**** HSC parameter indices ****
BASE_L16 .set 0
BASE_H16 .set 1
BWSCALE_L16 .set 2
BWSCALE_H16 .set 3
HOP_L16 .set 4
HOP_H16 .set 5
DCF_P .set 6
DCF_H16 .set 7
FCSPACE_L16 .set 8
FCSPACE_H16 .set 9
FLAGS_L16 .set 10
FLAGS_H16 .set 11
LOSTI_L16 .set 12
LOSTI_H16 .set 13
LOCOM_L16 .set 14
LOCOM_H16 .set 15
OFFSET_L16 .set 16
OFFSET_H16 .set 17
TIMEL_L16 .set 18
TIMEL_H16 .set 19
TIMEN_L16 .set 20
TIMEN_H16 .set 21

**** NCO constants and commands ****
NCO_INIT .float 1.92e5
NCO_const .float 1.0737418e2
D0_D7mask .word OFFFh
D0_D1mask .word OFFF0h
D16_D21mask .word OFFF000h
D14_D21mask .word OFFF0000h
D0_D7addr .word 3000000h
D0_D1addr .word 2000000h
D16_D21addr .word 1000000h
D24_D31addr .word 0h
Addr_Phase .word Phase
NCO_WRN_L0 .set 4h
NCO_STRB_HI .set 7h

**** Reserve memory in .bss for variables ****
globl op_mode
.bss op_mode,1
globl LSB16
.globl LSB16,1
.globl MBS16
.globl MBS16,1
.globl f_select
.globl f_select,1
.globl HOP_PAR
.globl HOP_PAR,1
.globl temp_var
.globl temp_var,1
.globl NCO_CLK
.globl NCO_CLK,1
.globl Phase
.globl Phase,4

B19
.globl phs_rnded
.globl phs_rnded.1
.globl original_phase
.globl original_phase.1
.globl hop_dwell
.globl hop_dwell.1
.globl MSC_array
.globl MSC_array.25
.globl counter
.globl counter.1
.globl tx_cnt
.globl tx_cnt.1
.globl gt_vars
.globl gt_vars.12

**** Variables defined elsewhere ****
.globl BER_stat
.globl hop_cnt
.globl frm_cnt
.globl prev_FRO
.globl chk_FRO_flg
.globl array_cnt
.globl hyp_used
.globl hyp_offset
.globl ntx_prb_frm
.globl act_prb_frm
.globl act_prb_hyp
.globl start_timef
.globl start_timeh
.globl assigned_f
.globl tms_sync_rdy
.globl brst0_flg
.globl brst1_flg
.globl user_hyp_off
.globl iter_hyp
.globl first_prb
.globl prb_cmd
.globl coarse1
.globl coarse2
.globl ref_frame

**** Initialize RESET and interrupt service routine locations ****

.sect "VECTORS"

RESET .word  START
INT0  .word  GT_ISR
INT1  .word  0
INT2  .word  0
INT3  .word  0
XINT0 .word  0
XINT1 .word  0
RINTA .word  0
RINT0 .word  0
RINT1 .word  0
DINT  .word  0
.space  20

**** Program begins here ****

.text

START:

LDI      INTPG,DP ; set DP pointer to 0
LDI      ST_reg_init,ST ; clear/enable CACHE, disable GVM
LDI      DATAPG,DP
LDI      @HEAPSTACK,SP ; initialize SW stack pointer

LDI      BUSPEC,DP ; initialize primary bus control register
LDI      Priusval.R4
STI      R4,@Priusbus
LDI      Secusval.R4 ; initialize secondary bus control register
STI      R4,@Secbus

LDI      DPLNKPG,DP

LDI      SW_RES_GT,RO ; issue SW reset of GT i/f board
LSH      16,RO
STI      RO,@COMMAND
CALL      DSPDLAYLP

LDI      Default_CMD,RO
LSH      16,RO
STI      RO,@COMMAND
CALL      DSPDLAYLP
LDI  BERSwres,RO ;issue SW reset of BERSw board
LDI  16,R0
LTH  R0,$CMD_BER
CALL  DSPFDLAYLP
LDI  BERSwswal,R0
LTH  16,R0
STI  R0,$CMD_BER
CALL  DSPFDLAYLP

* Download parameters for GT processor
CALL  Rd_GTparam

* Wait for clock frequency from PC
LDI  DPMEMPG,DP

NCO_WAIT:
LDI  $PHS_AVAIL,R0
CMPI  BITSET,R0
BNE  NCO_WAIT
LDI  BITCLR,RO
STI  R0,$PHS_AVAIL
LDI  $PHS_RNDED,R0
LDI  BRSFG,DP
STI  R0,$phs_rnded
STI  R0,$original_phase
LDI  BITSET,R0
LDI  DPMEMPG,DP
STI  R0,$PHS_ACK
CALL  START_NCO

* Download initial parameters for HSC (Blue box)
CALL  Rd_HSCparam
CALL  HSC_INIT
CALL  Ld_param_HSC

* Final initialization of variables
LDI  DPLNKPG,DP ;make sure RF is on
LDI  0,R0
STI  R0,$PSK_FRM
CALL  DSPFDLAYLP
LDI  DPLNKPG,DP
LDI  $STATUS,R0
CALL  DSPFDLAYLP
LDI  0,R0 ;clear flags for ISR
LDI  BRSFG,DP
STI  R0,$BER_stat
STI  R0,$chk_FRO_flg
STI  R0,$array_cnt
STI  R0,$msrcync_rdy
LDI  800h,R0 ;initialize prev_FRO to be "high"
STI  R0,$prev_FRO
LDI  222,R0 ;arbitrary
STI  R0,$chop_cnt
LDI  7,R0
STI  R0,$frm_cnt

* Enable interrupts
LDI  0,IF ;clear any pending interrupts
CALL  Enableint

NEXT_CMD:

* Wait for mode of operation from PC
LDI  DPMEMPG,DP

MODE_WAIT:
LDI  $MODE_AVAIL,R0
CMPI  BITSET,R0
BEQ  GET_MODE
LDI  $STOP_BNTR,R0
CMPI  BITSET,R0
BEQ  CLOSE_OUT
B  MODE_WAIT
GET_MODE:
LDI BITCLR, R0 ; clear MODE_AVAIL location
STI R0, @MODE_AVAIL
LDI @MODE, R0 ; read MODE
LDI BSFG, DP
STI R0, @op_mode
LDI BITSET, R0 ; acknowledge MODE transfer
LDI DPMEMPG, DP
STI R0, @MODE_ACK

* Put MSC in known state (STOP MODE)
CALL MSC_INIT

* Find out which mode
LDI BSFG, DP
LDI @op_mode, R0
Cmpi MODE1, R0
BEQ OP_1
Cmpi MODE2, R0
BEQ OP_2
Cmpi MODE3, R0
BEQ OP_3
Cmpi MODE4, R0
BEQ OP_4
Cmpi MODE5, R0
BEQ OP_5
Cmpi MODE6, R0
BEQ OP_6
Cmpi MODE7, R0
BEQ OP_7

OP_ERR:
LDI BSFG, DP ; Error. Synth goes to low edge of alloc BW
LDI 0h, R0
STI R0, @LSB16
LDI 0h, R0
STI R0, @MSB16
CALL CHG_COM_DISP
B RET_MODE

***********************
OP_1:
CALL CHOOSE_F
B RET_MODE
***********************

OP_2:
LDI DSPLNPSC, DP
CALL NRDY_low_loop
LDI CHG_HOP, R0 ; issue CHG_HOP to respond to rising edge of NCLK
LSH 16, R0
STI R0, @MSC_PORT
CALL DSPPLAYLF
CALL NRDY_low_loop
LDI RUN_MSC, R0 ; issue RUN command
LSH 16, R0
STI R0, @MSC_PORT
CALL DSPPLAYLF
B RET_MODE

***********************
OP_3:
DWELL_WAIT:
LDI DPMEMPG, DP
LDI @DWELL_AVAIL, R0
Cmpi BITSET, R0
BNE DWELL_WAIT
LDI BITCLR, R0
STI R0, @DWELL_AVAIL
LDI @DWELL_VAL, R0
LDI BSFG, DP
STI R0, @op_dwell
LDI BITSET, R0
LDI DPMEMPG, DP
STI R0, @DWELL_ACK

INCR_WAIT:
LDI DPMEMPG, DP
LDI @INCR_AVAIL, R0
Cmpi BITSET, R0
BNE INCR_WAIT

B22
LDI      BITCLR, R0
STI      R0, @INCR_AVAIL
LDI      @INCR_VAL, R0
LDI      BSSPG, DP
STI      R0, @Hop_incr
LDI      BITSET, R0
LDI      DPMEMPG, DP
STI      R0, @INCR_ACK
CALL     SLOW_MOP
B         RET_MODE

*****************************************************************************

OP_4:
FR0_DET:
LDI      BITSET, R0
LDI      BSSPG, DP
STI      R0, @chk_FR0_flag
B         RET_MODE

*****************************************************************************

OP_5:
RC_FR0:
LDI      BITCLR, R0
LDI      BSSPG, DP
STI      R0, @chk_FR0_flag
LDI      DPMEMPG, DP ;clear pending RIS_DET
STI      R0, @RIS_DET
B         RET_MODE

*****************************************************************************

OP_6:
CALL     CRSF_SYNC ;ONLY APPLIES TO MODE6 & MODE7
B         NEXT_CMD

*****************************************************************************

OP_7:
CALL     FINC_SYNC
B         NEXT_CMD

*****************************************************************************

RET_MODE:
LDI      DPMEMPG, DP
LDI      BITSET, R0
STI      R0, @C10DONE
B         NEXT_CMD

CLOSE_OUT:
CALL     Disable_int
LDI      DPMEMPG, DP
LDI      BITCLR, R0
STI      R0, @STOP_INT

STDBY:
MOP
B         STDBY ;C10 waits here

*****************************************************************************

*****************************************************************************

SUBROUTINES BEGIN HERE
*****************************************************************************

.globl   CHGC_CON_DISP

CHGC_CON_DISP:
PUSH     DP
PUSH     R0
PUSHF    R0
LDI      DSPKMPG, DP
CALL     NRDY_low_loop ;wait until HSC has processed command
LDI      CHGC_IMMED, R0
LSH      16, R0
STI      R0, HSC_PORT
CALL     DSPFLAYLP
CALL     NRDY_low_loop
LDI      LD_LATCH, R0 ;Load Latch command
LDI      LD_HOP, R0 ;Load Hop command
LSH      16, R0
STI      R0, HSC_PORT
CALL     DSPFLAYLP
CALL NRDY_low_loop
LDI BSSPG, DP
LDI $LSB16, RO ; 16 LSB of COMSTRN synthesizer latch
LSH 16, RO
LDI DEPLNKPG, DP
STI RO, #HSC_PORT
CALL DSPDLYLP
CALL NRDY_low_loop
LDI BSSPG, DP
LDI $MSB16, RO ; 16 MSB of COMSTRN synthesizer latch
LSH 16, RO
LDI DEPLNKPG, DP
STI RO, #HSC_PORT
CALL DSPDLYLP
CALL NRDY_low_loop
PPOP RO
PPOP RO
PPOP DP
RETS ; end of CHG_COM_DISP subroutine

.global CHOOSE_F

CHOOSE_F:
    PUSH DP
    PUSH RO
    PUSH R1
    PUSH R2
    PUSH R3
    PUSH R4
    PUSH R5
    PUSH R6
    PUSH R6

    * Wait for frequency selected by PC
    LDI DPMEMP, DP
    LDI $F_AVAIL, RO ; wait until frequency selection is available
    CMP RO, RO
    BNE
    F_WAIT
    LDI BITCLR, RO
    STI RO, $F_AVAIL
    LDI $F_VALUE, RO ; read F_VALUE
    LDI BSSPG, DP
    STI RO, $F_select
    LDI BITSET, RO
    STI RO, #F_ACK

    * Send appropriate frequency to HSC

    * First, check that it's not an F_STOP command
    LDI BSSPG, DP
    LDI $F_select, RO ; R0 <- f_select
    CMP RO, RO
    BEQ GET_OUT

    * For now, value to be transferred to HSC computed by PC
    AND MASKL16, RO ; store 16 LSB
    STI RO, $LSB16
    LDI $F_select, RO
    LSH -16, RO
    AND MASKL16, RO ; store 16 MSB
    STI RO, #MSB16
    CALL CHG_COM_DISP

B24
GET_CUT:
PPOF R6
POP R6
PPOF R5
POP R5
PPOF R4
POP R4
PPOF R3
POP R3
PPOF R2
POP R2
PPOF R1
POP R1
PPOF R0
POP R0
POP DP

;end of CHOOSE_F subroutine

*******************************************************************************
.globl COMMAND_CLK
.globl index
.bss index,1
.text

COMMAND_CLK:
PUSH DP
PUSH R0
PUSHF R0
PUSH R1
PUSHF R1
PUSH R2
PUSHF R2
PUSH ARO
PUSH IRO

LDI BSSPG,DP
LDI 0,R0
LDI R0,IRO
STI R0,#index

LDI DATAPG,DP
LDI @Addr_Phase,AR0

Phase_trans:
LDI **AR0(IRO),R1
LDI DISPFILEG,DP
STI R1,#NO_CMD
CALL DSPFILEYP
LDI NO_CMD,HI,R2
LSH 16,R2
STI R2,#COMMAND
CALL DSPFILEYP
LDI Default_CMD,R2
LSH 16,R2
STI R2,#COMMAND
CALL DSPFILEYP

LDI BSSPG,DP
LDI @index,R0
ADDI 1,R0

LDI R0,IRO
STI R0,#index

CMPI 4,R0
BGT Phase_trans

LDI DISPFILEG,DP
LDI NO_STRH_HI,R2
LSH 16,R2
STI R2,#COMMAND
CALL DSPFILEYP
LDI Default_CMD,R2
LSH 16,R2
STI R2,#COMMAND
CALL DSPFILEYP

POP IR0
POP ARO
POP R2
PPOF R2
POP R1
POP R1
PPOF R0
POP R0
POP DP

B25
RETS ;end of COMMAND_CLK subroutine

******************************************************************************
.globl DSPDLAYLP

DSPDLAYLP:
PUSH DP
PUSH R1
PUSHF R1
LDI 0,R1

DLAY:
ADDI 1,R1
CMPI 5,R1
BLT DLAY

POPF R1
POP R1
POP DP

RETS ;end of DSPDLAYLP subroutine

******************************************************************************
.globl HSC_INIT

HSC_INIT:
PUSH DP
PUSH R0
PUSHF R0

CALL NRDY_low_loop ;initialize HSC

LDI DSPLNKPG,DP

LDI STOP_HSC,R0 ;1st STOP command

LSH 16,R0
STI R0,#HSC_PORT

CALL DSPDLAYLP

CALL NRDY_low_loop ;wait until HSC has processed command

LDI STOP_HSC,R0 ;2nd STOP command

LSH 16,R0
STI R0,#HSC_PORT

CALL DSPDLAYLP

CALL NRDY_low_loop ;wait until HSC has processed command

LDI STOP_HSC,R0 ;3rd STOP command

LSH 16,R0
STI R0,#HSC_PORT

CALL DSPDLAYLP

POPF R0
POP R0
POP DP

RETS ;end of HSC_INIT subroutine

******************************************************************************
.globl Ld_param_HSC

Ld_param_HSC:
PUSH DP
PUSH R0
PUSHF R0
PUSH A0
PUSH I0

LDI DATAPG,DP

LDI #HSC_addr,A0

LDI DSPLNKPG,DP

CALL NRDY_low_loop ;wait until HSC has processed command

LDI LD_BASE,R0 ;Load BASE command

LSH 16,R0
STI R0,#HSC_PORT

CALL DSPDLAYLP

CALL NRDY_low_loop ;wait until HSC has processed command

LDI BASE_L16,I0 ;BASE_L16 transfer

LDI *+A0(I0),R0

B26
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI BASE_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI LD_BMSCL, R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI BMSCL_L16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI BMSCL_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI LD_DOPF, R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI DOPF_L16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI DOPF_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI LD_FCSPL, R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FCSPL_L16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FCSPL_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI LD_FLAGS, R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FLAGS_L16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FLAGS_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FLAGS_L16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP

CALL NRDFY_low_loop ; wait until HSC has processed command
LDI FLAGS_H16, R0
LDI +AR0(IRD), R0
LSH 16, R0
STI R0, #HSC_PORT
CALL DSEFPLAF LP
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LDLOGC, RO ;Load LOGC command
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOGC_L16, IR0 ;LOGC_L16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOGC_M16, IR0 ;LOGC_M16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOCOM, RO ;Load LOCOM command
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOCOM_L16, IR0 ;LOCOM_L16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOCOM_M16, IR0 ;LOCOM_M16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LOFFSET, RO ;Load OFFSET command
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI OFFSET_L16, IR0 ;OFFSET_L16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI OFFSET_M16, IR0 ;OFFSET_M16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI LO_TIMELO, RO ;Load TIMELO command, autocr up 32 bits (TIMEHI)
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI TIMELO_L16, IR0 ;TIMELO_L16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
CALL NRDY_low_loop ;wait until HSC has processed command
LDI TIMELO_M16, IR0 ;TIMELO_M16 transfer
LDI *+AR0(IR0), RO
LSH 16, RO
STI ROC, HSC_PORT
CALL DSPLALYP
POP IR0
POP AR0

B28
POPF          RO
POPF          RO
POPF          DP
RETS          ;end of Ld_param_HSC subroutine

                     .globl    NRDV_low_loop

NRDV_low_loop:
  PUSH          DP
  PUSH          RO
  PUSHF         RO
  LDI           DSPLNKPG, DP

NRDV_LOOP:
  LDI           $STATUS, RO
  LSH           -16, RO
  TSTB          NRDV_BIT, RO
  BNE           NRDV_LOOP
  POPF          RO
  POP            RO
  POP            DP
RETS           ;end of NRDV_low_loop subroutine

                                                                                     .globl    Rd_GTparam

Rd_GTparam:
  PUSH          DP
  PUSH          ARO
  PUSH          IRO
  PUSH          RO
  PUSHF         RO
  LDI           DATABPG, DP
  LDI           $VAR_BASE, ARO
  LDI           DPMEMPG, DP

WAIT_GT_VAR:
  LDI           $END_FILE, RO
               ;check for end of data file
  CMP           BITSET, RO
  BREQ          FINISHED
  CMP           $DAT_AVAIL, RO
               ;check for data available
  CMP           BITSET, RO
  BNE           WAIT_GT_VAR

READ_VAL:
  LDI           BITCLR, RO
  STI           RO, $DAT_AVAIL
  LDI           $INDEX, IRO
  LDI           $DAT_VALUE, RO
  CMP           RO, -*AR0(IRO)
  BNE           BD
  STI           RO, $DAT_ACK
  CMP           BITSET, RO
  BNE           WAIT_GT_VAR

FINISHED:
  LDI           BITCLR, RO
  LDI           DPMEMPG, DP
  STI           RO, $END_FILE
  POPF          RO
  POP            RO
  POP            IRO
  POP            ARO
  POP            DP
RETS           ;end of Rd_GTparam subroutine

                                                                                     .globl    Rd_HSCparam

Rd_HSCparam:
  PUSH          DP
  PUSH          RO
  PUSHF         RO
  PUSH          ARO
  PUSH          IRO
  LDI           DPMEMPG, DP

RD_NEXT:
  LDI           DPMEMPG, DP

WAIT_DAT:
  LDI           $VAL_RDY, RO
  CMPI          BITSET, RO
               ;check for next param ready

B29
REQ RD_DAT
LDA #$E9C111,RO ;also check for end of file
CMPF BITSET.RO
BNE WAIT_DAT
B WRAP_UP

RD_DAT:
LDI 0,RO ;clear VAL_RDY
STI R0,#VAL_RDY
LDI #VAL_IDX,IR0
LDI DPMEMP.RO
LDI #VALUE,RO
LDI #HSC_addr_AR0
SSP0,DP
STI R0,•AR0(IR10) ;store next param in array
LDI BITSET.R0
LDI DPMEMP.RO
STI R0,#VAL_ACK
B RD_NEXT

WRAP_UP:
PPOP IR0
POP AR0
PPOP R0
POP DP

RETS ;end of Rd_HSCparam subroutine

***********************************************************************
globl SLOW_HOP
.globl hop_num
.bss hop_num,1
.globl hop_incr
.bss hop_incr,1

.text

SLOW_HOP:
PUSH DP
PUSH RO
PUSHP RO
PUSH IOF
DO_AGAIN:
LDI 0,RO
LDI BSSEP,DP
STI R0,#hop_num

GO_THRU_PH:
LDI BSSEP,DP
LDI #hop_num,RO
AND MASK16.RO
STI RO,#LSRB16
LDI #hop_num,RO
LSH -16,RO
AND MASK16.RO
STI RO,#MSRB16
CALL CHG_COM_DISP
LDI BSSEP,DP
LDI XFO_EN,IOF ;enable XFO
OR XF_SET,IOF ;put "1" on XF
NOP
NOP
NOP
AND XFO_CLR,IOF ;put "0" on XF

HOLD:
RPTS #hop_dwell
NOP
OR XF_SET,IOF ;put "1" on XF
NOP
NOP
AND XFO_CLR,IOF ;put "0" on XF
NOP
NOP
NOP
OR XF_SET,IOF ;put "1" on XF
NOP
B30
NOP
AND  XF_CLR, IOF ; put "0" on XF

LDI  DPMENPG, DP
LDI  @STOP_MOD3, RO
CPFI  1, RO
BEQ  EXIT_LOOP

LDI  BSSPG, DP
LDI  @hop_num, RO
ADDI  @hop_incr, RO
STI  R0, @hop_num
LDI  DATAPG, DP
CPFI  @HSC_endhop, RO
BLC  GO_THRU_FN
B  DO_AGAIN

EXIT_LOOP:

LDI  DPMENPG, DP
LDI  BITCLR, RO
STI  R0, @STOP_MOD3

POP  IOF
PPOP  R0
POP  R0
POP  DP

RETS ; end of SLOW_HOP subroutine

*******************************************************************************

.globl START_NCO

START_NCO:
PUSH  DP
PUSH  R0
PUSHF  R0
PUSH  R1
PUSHF  R1
PUSH  R2
PUSHF  R2
PUSH  R3
PUSHF  R3

LDI  BSSPG, DP
LDI  @phs_rnded, RO
LDI  DATAPG, DP
AND  @D24_D3mask, RO
LSH  -8, RO
ADDI  @D24_D3addrs, RO
LDI  BSSPG, DP
STI  R0, @Phase

LDI  @phs_rnded, RO
LDI  DATAPG, DP
AND  @D16_D2mask, RO
LSH  8, RO
ADDI  @D16_D2addrs, RO
LDI  BSSPG, DP
STI  R0, @Phase+1

LDI  @phs_rnded, RO
LDI  DATAPG, DP
AND  @D8_D1mask, RO
LSH  8, RO
ADDI  @D8_D1addrs, RO
LDI  BSSPG, DP
STI  R0, @Phase+2

LDI  @phs_rnded, RO
LDI  DATAPG, DP
AND  @D0_D7mask, RO
LSH  16, RO
ADDI  @D0_D7addrs, RO
LDI  BSSPG, DP
STI  R0, @Phase+3

CALL  COMMAND_CLK

PPOP  R3
POP  R3
PPOP  R2
POP  R2
PPOP  R1
POP  R1
PPOP  R0
POP  R0

B31
POP    DF
RETS

; end of START_MCO subroutine

.end
B3. Coarse synchronization routine

Program Name: CSYNC.ASM
Author: C. Tom
Date edited: 30 March 1998
Description: Assembler code to be added to TMS_UL2.ASM which performs
crude synchronization. Based on state diagram approach
which directs path based on current state and triggers.

**** Subroutine declaration ****
.globl CSRE_SYNC
.globl CHG_STATE
.globl COMMAND_CLK
.globl DisableInt
.globl DSPULAYLP
.globl INPUT_HYP_LOG
.globl NRDY_low_loop
.globl RETRY_HYP
.globl START_MCO

**** Miscellaneous constants ****

XFO_EN .set 2h
XF_SET .set 6h
XF_CLR .set 0fffh
BITSET .set 1
BITCLR .set 0
STATE_ROW .set 8
STATE_COL .set 5
MASKL16 .set 0FFFFFFh

VAR_BASE .word gt_vars

* Indices for gt_vars parameters array
NUM_HOP .set 0
MAX_FRM .set 1
MAX_HOP .set 2
PRB_START .set 3
SRCH_LIM .set 4
TIMES_4_CONF .set 5
RESP_BUF_SIZE .set 6
MIN_DET_2_VER .set 7
NUM_RETRANSMITS .set 8

* TRIGGERS for CSYNC routine

CHD_RECD .set 0
PRELIM_COMPL .set 1
DET_RECD .set 2
DET_CONF .set 3
FASE_DET .set 4
PLINE_CLR .set 5
SRCH_RG_XCD .set 6
PLINE_CFG .set 7

* STATES for CSYNC routine

IDLE .set 0
PRELIM_INIT .set 1
GEN_PROBE .set 2
VER_DETECT .set 3
CLR_RESP_PIPE .set 4
GO_2_FSYNC .set 5
SRCH_EXCEED .set 6
PLINE_ERR .set 7

STAT_ADDR .data
HYP_LOG_ADDR .word
HYP_FRAME .word
RESP_BUF_ADDR .word
BITTS_M .word
BITTS_M1 .word
BITTS_M2 .word

B33
**** STATE LOOKUP TABLE ****

.data

;STATE Look up table
; ------- trigger
; | --- curr_state
; | /
; /

LK_UP_BASE:

.word SYNC_INIT 0,0=0
.word 0,1=1
.word 0,2=2
.word 0,3=3
.word 0,4=4
.word 0,5=5
.word 0,6=6
.word 0,7=7
.word 0,8=8
.word 0,9=9
.word 0,10=10
.word 0,11=11
.word 0,12=12
.word 0,13=13
.word 0,14=14
.word 0,15=15
.word 0,16=16
.word 0,17=17
.word 0,18=18
.word 0,19=19
.word 0,20=20
.word 0,21=21
.word 0,22=22
.word 0,23=23
.word 0,24=24
.word 0,25=25
.word 0,26=26
.word 0,27=27
.word 0,28=28
.word 0,29=29
.word 0,30=30
.word 0,31=31
.word 0,32=32
.word 0,33=33
.word 0,34=34
.word 0,35=35
.word 0,36=36
.word 0,37=37
.word 0,38=38
.word 0,39=39


**** Data page pointers ****

INITG .set 0
DPMEMPG .set 3h
BASS .set 60h
DATMG .set 0h
DPLINKPG .set 80h
RUSPG .set 80h
DBUPG .set 80h


**** DPMEM addresses ****

DPBASE .set 30000h

* DPBASE to DPBASE+18h used in TMS_UL2.asm

HYPAVAIL .set DPBASE+19h
USERHYP .set DPBASE+1Ah
HYPK .set DPBASE+18h
DBAVA .set DPBASE+1Ch
DBUC .set DPBASE+10h
DBUCACK .set DPBASE+18h
CHKAVAIL .set DPBASE+19h
CHKVAL .set DPBASE+20h
CHKACK .set DPBASE+21h
PMREF .set DPBASE+22h
CORESP .set DPBASE+23h
C1RESP .set DPBASE+24h
F0RESP .set DPBASE+25h
F1RESP .set DPBASE+26h
RESPAVAIL .set DPBASE+27h
RDU4RESP .set DPBASE+28h
RESPACK .set DPBASE+29h
XGND .set DPBASE+2Ah
CSYNC .set DPBASE+28h
LIQEND .set DPBASE+28h
PLINE_FLAG .set DPBASE+2Fh
**** GT i/f board addresses ****

COMMAND .set 800004h;WRITE only
STATUS .set 800004h;READ only
NCO_CMD .set 800005h;WRITE only
INTRPT_PORT .set 800005h;READ only
HSC_PORT .set 800006h;WRITE only
PSK_FRM .set 800007h;WRITE only

**** HSC commands and miscellaneous ****

HSC_endhop .data 0ffffffh;16777215 (max hop number for HSC)
SYNC_BIT .set 2;SYNC on bit D1 of GT status
NRDY_BIT .set 4;bit D2 on GT status (DSFLINK)
STOP_HSC .set 0
RDY_HSC .set 8000h
CHG_DNSED .set 0501h
CHG_HOP .set 0500h
LD_LATCH .set 0100h
LD_BASE .set 010Ch
LD_BSCALE .set 0106h
LD_DOFF .set 0110h
LD_FCSFACE .set 010Ah
LD_FLAGS .set 011Ah
LD_FKCHAN .set 0108h
LD_HOP .set 0104h
LD_LOSCI .set 0118h
LD_LOCOM .set 0112h
LD_OFFSET .set 0108h
LD_TIMELO .set 0114h
LD_TIMENI .set 0116h
ULSYNC_CMD .set 5000h;GO TO ULSYNC MODE, channel 1, bin 3
ULGO_BASE .set 2000h;BASE VALUE OF ULGO COMMAND FOR GSYNC

**** NCO constants ****

NCO_DELAY .data 13A14CFh;resulting phase incr to delay NCO clock

**** Reserve section in RAM block 0 for debug log ****

DEBUG_ADDR .word 809A00h

**** Reserve memory in .bss for variables ****

globl curr_state
.globl trigger
.globl hyp_used
.globl hyp_offset
.globl nxt_prb_frm
.globl act_prb_frm
.globl act_prb_frm
.globl act_prb_hop
.globl act_prb_hop
.globl start_timef
.globl start_timef
.globl start_timeh
.globl assigned_f
.globl tms_csnc_rdy
.globl tms_csnc_rdy
.globl brst0_flg
.globl brst0_flg
.globl brst1_flg
.globl brst1_flg
.globl user_hyp_off
.globl user_hyp_off
.globl iter_hyp
.globl iter_hyp

B35
**** Variables defined elsewhere ****
.globl hop_cnt
.globl frm_cnt
.globl gt_vars
.globl original_phase
.globl phs_rnded

********************************************************************

**** Program begins here ****
********************************************************************

.text

CRSE_SYNC:

********************************************************************

; STATE 1: PRELIM_INIT
********************************************************************

SYNC_INIT:
LDI BSSPG,DP ;update current state
LDI PRELIM_INIT,R0
STI R0,#curr_state
LDI DATAPG,DP
LDI #VAR_BASE,AR2 ;set up AR2

;Step 1: Compute precalc time required by MSC
LDI BSSPG.DP ;allow time for HSC to precompute
LDI @frm_cnt.R0 ;hop frequencies for CSYNC
ADDI 1,R0 ;start @ 2nd frame after current + 6% = 3rd frm
CMPR *AR2(MAX_FRM),R0;check if number exceeds max frame
BLE UNDER_LIM

OVER_LIM:
LDI *AR2(MAX_FRM),R1;rollover frame
ADDI 1,R1
SUBI R1,R0

UNDER_LIM:
STI R0, @start_time
sti R0, @nxt_prb_frm
LDI DATAPG.DP
TSTS SBITS2_M, R0;round to next multiple of 4
BR SET_TIME
;already a multiple of 4

NXT_MULT4:
LDI DATAPG.DP
AND SBITS2_31, R0 ;mask off 2 LSBs
ADDI 4, R0
CMPR *AR2(MAX_FRM), R0;check if number exceeds max frame
BLE LT_LIM

GT_LIM:
LDI *AR2(MAX_FRM), R1;rollover frame
ADDI 1, R1
SUBI R1, R0

LT_LIM:
LDI BSSPG.DP
STI R0, @start_time
STI R0, @nxt_prb_frm

;Step 2: set TIMEL0 and TIMEL1 on HSC due to precalc time

SET_TIME:

MVI *AR2(NUM_HOP), R0; change start_time in frames to hops
LDI BSSPG.DP ;MAKE SURE NUM_HOP IS 320!!!
STI R0, @start_time

LDI DSPLANKPG.DP ;send LD_TIMEL0 command
CALL NRDY_low_loop
LDI LD_TIMEL0, R0
LSH 16, R0
STI R0, HSC_PORT
CALL DSFPLAYLP

CALL NRDY_low_loop
LDI BSSPG.DP
LDI @start_time16, R0 ;send 16 LSBs of start_time
AND MASK16, R0
LSH 16, R0
LDI DSPLANKPG.DP
STI R0, HSC_PORT
CALL DSFPLAYLP

CALL NRDY_low_loop
LDI BSSPG.DP
LDI @start_time16, R0 ;send 16 MSBs of start_time
LDI DSPLANKPG.DP
STI R0, HSC_PORT
CALL DSFPLAYLP

;Step 2a: Initialize variables

LDI BSSPG.DP
LDI BITCLR.R0
STI R0, @hyp_used ;hyp_used = 0
STI R0, @assigned_f ;assigned_f = 0
STI R0, @brstf_flg ;brstf_flg = 0
STI R0, @brst1_flg ;brst1_flg = 0
STI R0, @hyp_offset ;hyp_offset = 0
STI R0, @exs_cyans_rdy ;exs_cyans_rdy = 0
STI R0, @valid_resp_flg ;valid_resp_flg = 0
STI R0, @new_stat ;new_state = 0
STI R0, @buf_idx ;buf_idx = 0
STI R0, @last_resp_crd ;last_resp_crd = 0
STI R0, @last_resp_in ;last_resp_in = 0
STI R0, @over_count ;over_count = 0
STI R0, @last_hyp_tested ;last_hyp_tested = 0
STI R0, @pipe_idx ;pipe_idx = 0
STI R0, @hyp_index ;hyp_index = 0

LDI 1, R1 ;init hyp_log and hyp_frame to 9999
LDI R1, IR1
LDI 9999, R0
LDI DATAPG.DP
LDI GHYP_LOG_ADDR, AR0
LDI GHYP_PRINTF_ADDR, AR1
LDI 127, RC ; set repeat counter
RTB ZRO_LOOP
STI R0, *AR0+(IR1)
STI R0, *AR1+(IR1) ; store '9999' in hyp_log and hyp_frame
LDI BPSPD, DP
LDI ULOG_BASE, R0
STI R0, LSP

WAIT_4_HYP:
LDI DPMEPFG, DP ; in case user wants to start at
LDI @HYP_AVAL, R0 ; other than PRB_START=288
CMPI BITSET, R0
BNE WAIT_4_HYP

GET_HYP:
LDI BITCLR, R0
STI R0, @HYP_AVAL
LDI USER_HYP, R0
LDI BPSPD, DP
STI R0, @user_hyp_off ; read user hypothesis offset
LDI BITSET, R0
LDI DPMEPFG, DP
STI R0, LSP
LDI BPSPD, DP
LDI @start_timef, R0
STI R0, @act_prb_frm ; act_prb_frm = start_timef

CALL INPUT_HYP_LOG ; store 1st hyp_offset (default 0) in hyp_log
LDI BPSPD, DP
LDI *+AR3(PRB_START), R0 ; PRB_START = default hop # to start synch probes
LDI @user_hyp_off, R1
ADDI R1, R0
STI R0, @first_prb ; starting point for binary search
STI R0, @act_prb_hop ; act_prb_hop = PRB_START+user_hyp_off

* Final check to see if adjustment of frame and hop number needed
* because of user hyp offset

CMPI *+AR3(NUM_HOP), R0 ; check if number exceeds max hop #
BLE TX_CS, TX_CS
SUBI *+AR3(NUM_HOP), R0 ; rollover hop
STI R0, @act_prb_hop
LDI @next_prb_frm, R0
ADDI 1, R0
CMPI *+AR2(MAX_FRM), R0 ; check if number exceeds max frm #
BLE BELOW_LIM

ABOV_LIM:
LDI *+AR2(MAX_FRM), R1 ; rollover frame
ADDI 1, R1
SUBI R1, R0
BELOW_LIM:
STI R0, @act_prb_frm

; at this point, HSC has been set up for coarse synch, switch to
; ULSYNC mode for HSC and check for SYNC line from status before
; proceeding.

; Step 3: Change HSC to ULSYNC mode

TX_CS:

CALL NRDY_low_loop
LDI ULSYNC_CMD, R0
LBN 16, R0
LDI DSPFHLKFG, DP
STI R0, @HSC_PORT ; send ULSYNC_CMD to HSC

CALL DSPFDLAYLP

WAIT_4_SYNC:

LDI DSPFHLKFG, DP
LDI @STATUS, R0
LBN -16, R0
CALL DSPFDLAYLP
TSTB SYNC_BIT, R0
EZ WAIT_4_SYNC

; Step 4: now, allow precalc time for HSC to finish calculations

PRECALC_WAIT:

LDI BPSPD, DP
LDI @frm_cnt, R0 ; wait until frm_cnt = (start_timef - 1)
LDI @start_timef, R1
SUBI 1, R1

B38
CALL PAST_HOP0
LDI @hop_cnt,R0
CMPI 0,R0
BEQ PAST_HOP0

;STATE 2: GEN_PROBES

COMPUT_HYP:

LDI BSSPG,DP ;update current state
LDI GEN_PROBES,R0
STI R0,@curr_state
LDI BSSPG,DP ;set flag for ISR
LDI BITSET,R0
STI R0,@mas_csync_rdy
LDI BSSPG,DP
LDI @hyp_used,R0 ;check if hyp transmitted
CMPI BITSET,R0
BNH CHK_FRM ;will branch on first iter/run
LDI BITCLR,R0
LDI BSSPG,DP
STI R0,@hyp_used ;clear flag for hypothesis used
STI R0,@assigned_f ;clear flag for ISR
STI R0,brst0_f1g ;clear flag for brst0 probes (ISR)
STI R0,brst1_f1g ;clear flag for brst1 probes (ISR)
LDI ULOO_BASE,R0 ;reset prb_cmd
STI R0,@prb_cmd
B CONT_SRCH ;TEST ONLY

LDI BSSPG,DP ;check if search range is exceeded
ABS @hyp_offset,R0
CMPI "AR2(SRCH_LIM),R0
BLT CONT_SRCH

; Trigger = SRCH_R0_XCD, GO TO STATE 6 (SRCH_EXCEED)

CONT_SRCH:

LDI BSSPG,DP
LDI SRCH_R0_XCD,R0
STI R0,@trigger
CALL CHNG_STATE
LDI BSSPG,DP
LDI @new_state,R2
B R2

KEEP_FRM:

STI R0,@next_prb_fram
SUBI R1,R0

NEXT_HYP:

LDI BSSPG,DP
LDI @hyp_offset,R0
CMPI 0,R0
BLE NEXT_HYP_POS

NEXT_HYP_POS:

NEGI R0,R1 ;negate hypothesis offset
LDI BSSPG,DP
STI R1,@hyp_offset

CALL INPUT_HYP_LOG ;store hyp_offset in HYP_LOG & HYP_FRAME

LDI BSSPG,DP
LDI @first_prb,R2 ;compute actual hop # to tx
ADDI R1,R2 ;csync probe
STI R2,@act_prb_hop
B CHK_FRM

B39
NXT_HYP_POS:
NEGI R0,R1 ; negate hypothesis offset
ADDI 1,R1 ; incr hypothesis
LDI BSSPG,DP
STI R1,hyp_offset
CALL INPUT_HYP_LOG ; store hyp_offset in hyp_log and hyp_frame
LDI BSSPG,DP
LDI #first_prb,R2 ; compute actual hop # to tx
ADDI R1,R2 ; sync probe, adj frm # if
STI R2,act_prb_hop ; necessary
CMPI /*AR2(NUM_HOP),R2; check if number exceeds max hop */
BLT CHK_FRM
SUBI /*AR2(NUM_HOP),R2; rollover hop
STI R2,act_prb_hop
ADDI 1,R2
CMPI /*AR2(MAX_FRM),R2; check if number exceeds max frm */
BLE VALID_FRM
LDI /*AR2(MAX_FRM),R1; rollover frame
ADDI 1,R1
SUBI R1,R2
VALID_FRM:
STI R2,act_prb_frm

; up to here act_prb_hop & act_prb_frm have been calculated
; Now, go into loop of generating probes and collecting responses

CHK_FRM:
LDI BSSPG,DP
LDI @assigned_f,R0 ; check if already in assigned frame
CMPI BITSET,R0
BEQ CHK_RET_LNK
LDI @act_prb_frm,R0
LDI @frm_cnt,R1
CMPI R0,R1
BNE CHK_RET_LNK ; check if current is assigned frame

SET_TXPRB:
LDI BITSET,R0 ; set flag to signal assigned frame
STI R0,assigned_f

CHK_RET_LNK:
LDI DPMEMP,DP
LDI @RESP_AVAIL,R0
CMPI BITSET,R0
BNE COMPUT_HYP ; go back to top of loop

RD_RESP:
LDI DPMEMP,DP
LDI BITCLR,R0 ; clear flag
STI R0,@RESP_AVAIL
LDI @FRM_REF,R0 ; store sync response
LDI @CO_RESP,R1
LDI @CHI_RESP,R2
LDI BSSPG,DP
STI R0,ref_frame ; coarse sync response for ch0
STI R2,coarse1 ; coarse sync response for chi
LDI BITSET,R0 ; ack sync response
LDI DPMEMP,DP
STI R0,@RESP_ACK

; Check if response frame # is a multiple of 4 (probe every 4 frame - user allocation assumed.)
; Check also if frame # is later than the start_time -> set flag

LDI BSSPG,DP
LDI @ref_frame,R0
LDI DATAP,DP
TESTB @BITO0_N1,R0
BNE REST_WRAP_UP ; not a mult of 4, disregard response
LDI BSSPG,DP ; check if past starting point of probe frames
LDI @valid_resp_flg,R3
CMPI BITSET,R3
BEQ TEST_RESP
LDI BSSPG,DP
LDI @ref_frame,R0

B40
LDI @start_timef,R3 ;R3 - R0 = start_timef - ref_frame
CMP R0,R3
BGT RESP_WRAP_UP ;invalid response, disregard

LDI BITSET,R3
STI R1,#valid_resp_flg

; Current user allocated chl for coarse synch probes.
; See USYNC_CMD description above

TEST_RESP:
CMPI 0,R2 ;R2 <- coarsel
BEQ RESP_WRAP_UP ;no "DETECT" for either burst

; Trigger = DET_RECD, SAVE CURRENT HYP/PROBE SETTINGS, GO TO STATE 3 (VER_DETECT)

LDI BSSPG,DP ;save current hypothesis for resume probing
LDI @hyp_offset,R0
STI R0,#last_hyp_tested

LDI ULG_BASE,R0 ;reset prb_cmd
STI R0,#prb_cmd

LDI BITCLR,R0 ;clear variables related to tx of probes
STI R0,#hyp_used
STI R0,#assigned_f
STI R0,#brstl_flg
STI R0,#brstl_flg

LDI BSSPG,DP

LDI DET_RECD,R0 ;set trigger
STI R0,#trigger
CALL CHG_STATE
BSSPG,DP
LDI @new_state,R2
R2 ;go to STATE 3

RESP_WRAP_UP:
B COMPUT_HYP

; STATE 3: VER_DETECT

CHG_DET:
LDI BSSPG,DP ;modify current state
LDI VER_DETECT,R0
STI R0,#curr_state

LDI @last_resp_in,R0 ;init buffer index for response pipeline
STI R0,#buff_idx

; initialize variables for verifying detect

LDI BSSPG,DP
LDI BITCLR,R0
STI R0,#re_tx_cnt
STI R0,#det_count
STI R0,#end_count
STI R0,#EDN_count

CALL RETRV_HYP ;retrieve hypothesis to verify

LDI BSSPG,DP
LDI @nxt_prb_frm,R0 ;compute next frame for probe
ADDI 4,R0
CMPI @frm_cnt,R0 ;check if nxt_prb_frm is already passed
BLE FRM_NO_CHG

ADDI NXT_2_PRB
ADDI 4,R0 ;go to next multiple of 4

NXT_2_PRB:
CMPI **#AR2(MAX_FRM),R0 ;check if number exceeds max frm 
BLE FRM_NO_CHG
LDI **#AR2(MAX_FRM),R1
ADDI 1,R1
SUBI R1,R0

FRM_NO_CHG:
STI R0,#nxt_prb_frm
STI R0,#act_prb_frm
STI R0,#start_timef ;used to find when synch resp are valid

LDI @hyp_offset,R0 ;compute act_prb_hop with offset
LDI @first_prb,R1 ; first_prb includes user offset @ beginning
ADDI R0,R1
STI R1,act_prb_hop
CMPI @AR2 (NULL_HOP),R1; check if number exceeds max hop #
BLT REPEAT_CHK
SUBI @AR2 (NULL_HOP),R1; max hop exceeded, roll over
STI R1,act_prb_hop
LDI @nxt_prb_frm,R1 ; adjust frm# because of roll over
ADDI 1,R1
CMPI @AR2 (MAX_FRM),R1; check if number exceeds max frm #
BNE FNM_OK
LDI @AR2 (MAX_FRM),R2
ADDI 1,R2
SUBI R2,R1
FNM_OK:
STI R1,act_prb_frm

; At this point, act_prb_hop and act_prb_frm have been calculated for VER_DETECT

REPEAT_CHK:
LDI @BSPPG,DP
LDI @hyp_used,R0
CMPI BITSET,R0
BNE WAIT_4_FRM
LDI @BSPPG,DP
LDI BITCLR,R0
STI R0,@hyp_used
STI R0,@assigned_f
STI R0,@brst0_flg
STI R0,@brst1_flg
LDI @ULC_BASE,R0
STI R0,@prb_cnd
LDI @BSPPG,DP
LDI @nxt_prb_frm,R0
ADDI 4,R0
CMPI @AR2 (MAX_FRM),R0; check if number exceeds max frm #
BLE STORE_FRM
LDI @AR2 (MAX_FRM),R1; rollover frame
ADDI 1,R1
SUBI R1,R0
STORE_FRM:
STI R0,@nxt_prb_frm
LDI @act_prb_frm,R0 ; compute next probe frame
ADDI 4,R0 ; using act_prb_frm to avoid recalc
CMPI @AR2 (MAX_FRM),R0; act_prb_hop should be the same
BLE FRM_UNDER ; done separately in case of straddling
LDI 4,R0 ; of frame boundaries bew nxt_prb_frm
ADDI 1,R1
SUBI R1,R0 ; and act_prb_frm
FRM_UNDER:
STI R0,@act_prb_frm

WAIT_4_FRM:
LDI @BSPPG,DP
LDI @assigned_f,R0
CMPI BITSET,R0
BEQ LOOK_4_RESP
LDI @act_prb_frm,R0
LDI @frm_cnt,R1
CMPI R0,R1
BNE LOOK_4_RESP

GO_PROBE:
LDI @BSPPG,DP
BITSET,R0
STI R0,@assigned_f

LOOK_4_RESP:
LDI @DPMEMP,DP
LDI @RESP_AVAIL,R0
CMPI BITSET,R0
BNE REPEAT_CHK

GET_RESP:
LDI @BSPPG,DP
BITCLR,R0 ; clear flag
STI R0,@RESP_AVAIL
LDI @frm_RESP,R0
LDI @CO_RESP,R1
LDI @C1_RESP,R2
LDI @BSPPG,DP
STI R0,@ref_frame
STI R1,@coarse0
STI
LDI
LDI
STI
   R2,#coarse1
   BITSET,R3
   DPMEMP,D.P
   R3,#RESP_ACK

; Check response is a mult of 4 and references frame later than start_timef

LDI
LDI
LDI
TSTB
BNZ
   BSSPG,D.P
   @ref_frame,R0
   DATAP,D.P
   BITSO_N1,R0
   CLOSE_RESP
   ; look at lower 2 bits to see if a mult of 4
   ; not a mult of 4, disregard response

; FOR TEST ONLY

B     LOOK_4_DET
   ; test - go directly to testing response

; LDI
LDI
LDI
LDI
BLK
SAV_2_PIPE:
   BSSPG,D.P
   @ref_frame,R0
   ; check if frame > start_timef
   @start_timef,R1
   ; for valid response
   CMPI
   R0,R1
   ; R1 - R0 = start_timef - ref_frame
   BLK
   LOOK_4_DET
   ; valid verify response
   ; Otherwise,

B     BSSPG,D.P
   @buff_idx,R0
   ; check to see if anymore room in response
   CMPI
   @last_resp_clrd,R0
   ; pipeline buffer
   BNE
   ASSMNL_RESP

; ///////////////////////////////////////////////////////////////////////////////////////////
; Trigger = PLINE_OFLO, go to state 7 (OFLO_RESP)   ///////////////////////////////////////////////////////////////////////////////////////////

LDI
LDI
STI
CALL
LDI
LDI
ASSMNL_RESP:
   BSSPG,D.P
   PLINE_OFLO,R0
   RO,#trigger
   CHG_STATE
   BSSPG,D.P
   @new_state,R2
   B

; ///////////////////////////////////////////////////////////////////////////////////////////
; save responses arriving after "detect"
LDI
LDI
LSH
LSH
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
ADDI
CMPI
SUBI
SAV_BUF_IDX:
STI
B
   @ref_frame,R0
   @coarse0,R1
   @coarse1,R2
   16,R0
   2,R2
   R2,R0
   R1,R0
   DATAP,D.P
   001552140,AR0
   BSSPG,D.P
   @buff_idx,R3
   R3,1R0
   R0,~AR0(1R0)
   1,R3
   **AR2(RESERVED,SIX),R3
   **AR2(RESERVED,SIX),R3
   R3,0buff_idx
   CLOS.RESP

; See if "detect" received again
; Again, chl for current user assumed

LOOK_4_DET:

LDI
LDI
ADDI
STI
LDI
ADDI
STI
LDI
CMP
BNE
INCR_TALLY
B
      BSSPG,D.P
      @ref_rx_cnt,R0
      1,R0
      #ref_rx_cnt
      BSSPG,D.P
      @coarse1,R2
      0,R2
      INCR_TALLY
      ; "detect" recd again, incr counter

; no detect received, so check if # times retransmit completed

B

INCR_TALLY:

LDI
BSSPG,D.P

; increase # of "detect"s count
LDI @det_count,R0
ADDI 1,R0
STI R0, @det_count

CMPI 1,R2
BNE NOT_ND
LDI @ND_count,R0
ADDI 1,R0
STI R0, @ND_count
B CHK_NUM_DET

NOT_ND:
CMPI 2,R2
BNE ITS_DD
LDI @DN_count,R0
ADDI 1,R0
STI R0, @DN_count
B CHK_NUM_DET

ITS_DD:
LDI @DD_count,R0
ADDI 1,R0
STI R0, @DD_count

CHK_NUM_DET:
LDI BSFG,DP
LDI @det_count,R0
CMPI "AR2(MIN_DET2,VER),R0
BLT CHK_NUM_RETX
;if not enough yet, check if # retransmit completed
;otherwise,

;////////////////////////////////////////////////////////////
; Trigger = DET_CONFIRM, GO TO STATE 4 (GO_2_FSTNC) (for now, RETURN)
;////////////////////////////////////////////////////////////

;FOR TEST, GO STRAIGHT TO CHANGE:

B CHANGE

; first compare ND_count with both DD_ and DN_count, if ND > DD and DN, then
; have to delay NCO by 180 (see synch response scenarios)

LDI BSFG,DP
LDI @ND_count,R0
CMPI @DD_count,R0
BLT CHANGE
;if ND < DD, then no need to adjust clock

LDI BSFG,DP
LDI @DD_count,R0
CMPI @DN_count,R0
BLT CHANGE
;if ND < DN, then no need to adjust clock

DLY_B_180:
LDI DATAPG,DP
LDI @NDCO_DLY,R0
LDI BSFG,DP
STI R0, @phs_rnded
CALL START_NCO
;change to slower frequency

LDI BSFG,DP
LDI @hop_crt,R0
LDI @frm_crt,R1
STI R0, @hop_ref
STI R0, @frm_ref
;save current hop and frame #

WAIT_320_HOPS:
LDI BSFG,DP
LDI @hop_crt,R0
CMPI @hop_ref,R0
BNE WAIT_320_HOPS
LDI @frm_crt,R0
CMPI @frm_ref,R0
BEQ WAIT_320_HOPS
;frm # has not advanced (<320hops), keep waiting

LDI BSFG,DP
LDI @original_phase,R0
STI R0, @phs_rnded
CALL START_NCO
;change back to original NCO frequency

CHANGE:
LDI BSFG,DP
STI R0, @trigger
CALL CHG_STATE
LDI BSFG,DP
LDI @new_state,R2
B R2
;go to new state (state 4)

;check if hypothesis has been retransmitted enough times to determine whether to continue

B44
CHK_NUM_RETX:
LDI BSSPG,DP
LDI @re_tx_cnt,R0
CMPX "AR2(NUM_RETRANSMTS),R0
BLT CLOSE_RESP
;otherwise, finish all retransmits and not enough "detects" received to confm
;///////////////////////////////////////////////////////////////////////////////////
; Trigger = FALSE_DET, GO TO STATE 5 (CLR_RESP_PIPE)
;///////////////////////////////////////////////////////////////////////////////////

LDI BSSPG,DP ;save loc of last entry into resp_buff
LDI @buff_idx,R0
STI R0,@last_resp_in

LDI UG0_BASE,R0 ;reset prb_cmd
STI R0,prb_cmd

LDI BITCLR,R0 ;clear variables related to tx of probes
STI R0,hypr_used
STI R0,assigned_f
STI R0,brst0_flg
STI R0,brst1_flg

LDI BSSPG,DP ;set trigger
LDI FALSE_DET,R0
STI R0,trigger
CALL CHO_STATE
LDI BSSPG,DP
LDI @new_state,R2
B R2 ;go to new state (state 5)

CLOSE_RESP:

B REPEAT_CHK

; State 4: GO_2_FSYNC (temporarily set to RETURN)
;******************************************************************************

RETURN:

LDI BSSPG,DP
LDI GO_2_FSYNC,R0 ;modify current state
STI R0,@curr_state

LDI BITCLR,R0 ;clear flags for ir
STI R0,hypr_used
STI R0,assigned_f
STI R0,brst0_flg
STI R0,brst1_flg

LDI @hop_cnt,R0 ;adjust hop and frame counter
ADDI @hp_offset,R0
STI R0,hop_cnt
CMPI "AR2(MEM_HOP),R0;check for hop & rollover
BLE OK_2_CONTINUE
SUBI "AR2(MEM_HOP),R0
STI R0,hypr_cnt
LDI @frm_cnt,R0
ADDI 1,R0
CMPI "AR2(MAX_FRM),R0;check for frm & rollover
BLE LEAVE_FRM
LDI "AR2(MAX_FRM),R1
ADDI R1,R1
SUBI R1,R0

LEAVE_FRM:
STI R0,@frm_cnt

OK_2_CONTINUE:

LDI BITSET,R0
LDI DPMENPG,DP
STI R0,CSYNC_OK

RETS ;return to TMS_UL2.ASM

******************************************************************************
; State 5: CLR_RESP_PIPE
******************************************************************************

CLR_PIPE:

LDI BSSPG,DP ;revise current status
LDI CLR_RESP_PIPE,R0
STI R0,@curr_state

LDI DATAPG,DP

B45
LDI @RESP_BUF_ADDR, A0 ; set up indirect addr pointer, A0
LDI @last_resp_clrd, R0 ; last loc data cleared
LDI @last_resp_in, R1 ; last loc data entered
CMP R0, R1
BEQ PLINE_MT
STI R0, @pipe_idx ; initialize pipeline index

FLUSH_OUT:
LDI BSSPGDP
LDI @pipe_idx, R0 ; update pipeline index
ADDI 1, R0
CMPI *[AR2(RESP_BUF_SIZE), R0] ; check for pipe index rollover
BLT SAVE_PIPE_IDX
SUBI *[AR2(RESP_BUF_SIZE), R0]

SAVE_PIPE_IDX:
STI R0, @pipe_idx
LDI @pipe_idx, R1
LDI *[AR0(READ), R0] ; look at next response in pipeline
LDI DATAPG, DP
TSRB @bit52_M_1, R0 ; look at chl coarse synch response
BZ FLUSH_NXT
; no "DETECT" in response

; //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
; Trigger = DET_RECV, GO TO STATE 3 (GEN_PROBES) TO RESUME PROBING
; //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

LDI BSSPGDP
LDI @pipe_idx, R0 ; find out if any more to flush
CMPI @last_resp_in, R0 ; pipe_idx - last_resp_in
BNE FLUSH_OUT
; if pointers don't line up, more to flush
; otherwise, pipeline empty

FLUSH_NXT:
LDI BSSPGDP
LDI @pipe_idx, R0 ; find out if any more to flush
CMPI @last_resp_in, R0 ; pipe_idx - last_resp_in
BNE FLUSH_OUT
; if pointers don't line up, more to flush
; otherwise, pipeline empty

PLINE_MT:
LDI BSSPGDP
LDI ULGO_BASE, R0 ; reset prb_cmd
STI R0, @prb_cmd
LDI @nxt_prb_frm, R0 ; compute next frame to tx probes
ADDI 4, R0
CMPI @frm_cnt, R0
BGT NXT_FRM_1_PRB ; next frame number is valid
B TRY_NXT_FRM
; need to go to next allocation

NXT_FRM_1_PRB:
CMPI *[AR2(MAX_PRM), R0] ; check if frame # exceeded
BLE SAVE_PRM
LDI *[AR2(MAX_PRM), R1] ; rollover frame
ADDI 1, R1
SUBI R1, R0

SAVE_PRM:
STI R0, @nxt_prb_frm
STI R0, @act_prb_frm
STI R0, @start_timef

GEN_PROBES:
LDI @last_hyp_tested, R0 ; recover last hypothesis tested in

STORE_HOP:
STI R0, @act_prb_hop
LDI @nxt_prb_hop
ADDI 1, R0

B46
CMPI  +ARZ(MAX_FRM), R0; check if frame # exceeded
LBE  FRM_STORE
LDI  +ARZ(MAX_FRM), R1; rollover frame
ADDI  1, R1
SUBL  R1, R0
FRM_STORE:
STI  R0, &act_prb_frm

STORE_HYP:
LDI  DATARP, DP
LDI  @HYP_LOG_ADDR, AR0
LDI  BSSPG, DP
LDI  @next_prb_frm, IR0
LDI  @hyp_offset, R0
STI  R0, +AR0(1R0); store hyp_offset in hyp_log rel to frame #
LDI  PLINE_CLR, R0
STI  R0, Trigger
CALL  CHK_STATE
LDI  BSSPG, DP
LDI  @new_state, R2
B  R2; go to new state (state 2)

; State 6: SKCH_EXCEED

OUT_A_RANGE:
LDI  BSSPG, DP
LDI  SKCH_EXCEED, R0
STI  R0, @curr_state
LDI  DPHMXP, DP
STI  R0, @RNG_XCED

RETS  ;return to TMS_UL2.ASM (to replace WAIT_LOOP)

; State 7: PLINE_ERR (set flag to PC)

OFLO_RESP:
LDI  BSSPG, DP
LDI  PLINE_ERR, R0
STI  R0, @curr_state
LDI  DPHMXP, DP
STI  R0, @PLINE_FLAG

RETS  ;return to TMS_UL2.ASM

***************

CHG_STATE
***************

global

CHG_STATE:
FUSH  DP
FUSH  R0
FUSHF  R0
FUSH  R1
FUSHF  R1
FUSH  R2
FUSHF  R2
FUSH  ARO
FUSH  IR0

LDI  BSSPG, DP
LDI  @trigger, R0
LDI  @STATE_CNT, R1
MPFI  R0, R1
ADDI  @curr_state, R1
LDI  R1, IR0
LDI  DATARP, DP
LDI  @STAT_ADDR, AR0
LDI  +AR0(1R0), R2
LDI  BSSPG, DP
STI  R2, @new_state

POP  IR0
POP  ARO
POPF  R2
POP  R2

B47
PCPF  R1
POP   R1
PCPF  R0
POP   R0
POP   DP

RETS

******************************************************************************
*.globl   INPUT_HYP_LOG
******************************************************************************

INPUT_HYP_LOG:

PUSH   DP
PUSH   IR1
PUSH   AR0
PUSH   AR1
PUSH   AR3
PUSH   R0
PUSHF  R0

LDI    BSSPG,DP
LDI    @hyp_index,R0
LDI    R0,IR1

LDI    DATAPG,DP
LDI    @HYP_LOG_ADDR,AR0
LDI    @HYP_FRM_ADDR,AR1
LDI    `*AR0(IR1),R0
CMPF   9999,R0
BNE    HYP_BUFF_OFLO

LDI    BSSPG,DP
LDI    @next_prb_frm,R0
STI    R0,`*AR1(IR1)
LDI    @hyp_offset,R0
STI    R0,`*AR0(IR1)

ADDI   1,IR1
CMPF   128,IR1
BLT    SAV_HYP_IDX
LDI    BITCLES,IR1

SAV_HYP_IDX:

STI    IR1,@hyp_index
B      END_ROUTINE

HYP_BUFF_OFLO:

LDI    BITSET,R0
LDI    DPMEPG,DP
STI    R0,`WTOO_MANY_HYPS

END_ROUTINE:

POPF   R0
POP    R0
POP    AR3
POP    AR1
POP    AR0
POP    IR1
POP    DP

RETS

******************************************************************************
*.globl   RETRV_HYP
******************************************************************************

RETRV_HYP:

PUSH   DP
PUSH   IR0
PUSH   AR0
PUSH   AR3
PUSH   R0
PUSHF  R0

FIND_IDX:

LDI    DATAPG,DP
LDI    @HYP_FRM_ADDR,AR0
LDI    0,IR0

FIND_LOOP:

LDI    `*AR0(IR0),R0
LDI    BSSPG,DP
CMPF   @ref_frame,R0
BEQ    FOUND_IDX
ADDI   1,IR0

B48
LDI IRO,RO
CMPI 128,RO
BLT FIND_LOOP

LDI BITSET,RO ;set flag to PC, frm not found
LDI DPMEMPC,DP
STI R0,6FRM_NOT_FOUND

E CLOSE_ROUTINE

FOUND_IDX:
LDI DATAPG,DP ;IRO currently has index
LDI &HYP_LOG_ADDR,AR0
LDI **AR0(IRO),RO ;retrieve hyp_offset
LDI BSSPG,DP
STI R0,6hyp_offset

LDI 9999,RO
STI R0,**AR0(IRO) ;reset hyp_log location
LDI DATAPG,DP
LDI &HYP_FRM_ADDR,AR0
STI R0,**AR0(IRO) ;reset hyp_frame location

CLOSE_ROUTINE:
POPF R0
POP R0
POP AR0
POP IRO
POP DP

RETS
.end
B4. Fine synchronization routine

******************************************************************************

Program Name: FSYNC.ASM
Author: C. Tom
Date edited: 31 March 1998
Description: Assembler code to be added to TMS_UL2.ASM which performs
fine synchronization. Fine synchronization probes are
transmitted for user 1 (burst of 32). Fine sync responses
are analyzed in PC program. PC will take an average
of X' synch responses and compute an appropriate phase
change for the NCO. The phase change is such that the
adjustment of hop clock over 320 hops (1 frame). After
320 hops, the NCO is returned to its original frequency.
Fine synchronization occurs when fine sync response falls
below a certain threshold (i.e. fsync reap < threshold 'Y')
Currently considering threshold to be within 10% of a hop
******************************************************************************

**** Subroutine declaration ****
.globl FINE_SYNC
.globl CHG_FSTATE
.globl COMMAND_CLK
.globl DSPPLAYLF
.globl NRDY_low_loop
.globl START_NCO

**** Miscellaneous constants ****
XFO_EN .set 2h
XF_SET .set 6h
XF_CLK .set OFFFH
BITSET .set 1
BITCLR .set 0
MASKL16 .set OFFFH
NUM_ROW .set 6 ;number of rows in state table
NUM_COL .set 6 ;number of columns in state table
.data
BIT0_M_1 .word 3h
BIT52_31 .word OFFFFFFCh
VAR_BASE .word gt_vars
STATE_TBL .word STATE_BASE

**** TRIGGERS for FSYNC routine ****
CMD_ISSUED .set 0 ;starting point of FSYNC routine
FINISH .set 1 ;finished initialising variables, etc...
F_EST_AVAL .set 2 ;fine estimate available from PC
ADJ_COMPL .set 3 ;adjustment of NCO completed
HV_CONVING .set 4 ;no convergence of estimate after X times
CONVERGED .set 5 ;fine sync to within 10% of a hop achieved

**** STATES for FSYNC routine ****
IDLE .set 0 ;idle state
INIT_SECTION .set 1 ;performing preliminary initialisation
TX_PFROB .set 2 ;generating fsync probes and waiting for fine est
ADJ_NCO .set 3 ;adjusting NCO frequency over 320 hops
FINE_ping_ACH .set 4 ;not able to achieve fine sync, send error msg
GO_2_RUN .set 5 ;fine sync achieved, ready to send data

**** Look up table for FSYNC routine ****

; trigger 
; current state
;
; /
; /

STATE_BASE: .word FINE_INIT ;[0,0]=0
    .word [0,1]
    .word [0,2]
    .word [0,3]
    .word [0,4]
    .word [0,5]
    .word [1,0]=6
    .word FINE_PRB ;[1,1]
    .word [1,2]
    .word [1,3]
    .word [1,4]
    .word [1,5]

B50
.word 0 ;
.word 0 ;
.word CHANGE_NCO ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word FINE_PREB ;
.word NO_CONV ;
.word FINE_ACM ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;
.word 0 ;

**** Indices for gt_vars parameters array ****

MEM_HOP .set 0
MAX_FRM .set 1
MAX_HOP .set 2
PRB_START .set 3
LIM_10 .set 9
MIN_4_CONV .set 10
MAX_ATTEMPTS .set 11

**** Data page pointers ****

INTPG .set 0
DPMEMPG .set 3h
BSSTPG .set 80h
DATAPG .set 0h
DSPMPG .set 80h
BUSPG .set 80h
DBUGPG .set 80h

**** DPMEM addresses ****

DFBASE .set 30000h
FSTART_AVAIL .set DFBASE + 40h
FSTART_FRM .set DFBASE + 41h
FSTART_ACK .set DFBASE + 42h
EST_AVAIL .set DFBASE + 43h
FINE_EST .set DFBASE + 44h
PHS_CHANGE .set DFBASE + 45h
EST_ACK .set DFBASE + 46h
NO_FSYNC .set DFBASE + 47h
FSYNC_OK .set DFBASE + 48h

**** GT i/f board addresses ****

COMMAND .set 800004h ;READ only
STATUS .set 800004h ;READ only
NCO_CMD .set 800005h ;READ only
INTRPT_PORT .set 800005h ;READ only
HSC_PORT .set 800006h ;WRITE only
FSK_FRM .set 800007h ;WRITE only

**** HSC commands and miscellaneous ****

HSC_endhop .data
.word 0ffffffh ;16777215 (max hop number for HSC)
SYNC_BIT .set 2 ;SYNC on bit D1 of GT status
NRDY_BIT .set 4 ;bit D2 on GT status (DSPLINK)

STOP_HSC .set 0
NOS_HSC .set 8000h
CHG_IMMED .set 0501h
CHG_HOP .set 0500h
LD_LATCH .set 0300h
LD_BASE .set 030Ch
LD_BSCALE .set 0306h
LD_DCNF .set 0310h
LD_FCSPACE .set 030Ah
LD_FLAGS .set 031AH
LD_FSYNC .set 0308h
LD_HOP .set 0304h

B51
LD_LOSCI .set 0118h
LD_LOCOM .set 0112h
LD_OFFSET .set 00Dh
LD_TIMELO .set 0114h
LD_TIMEDI .set 0116h
ULSYNC_CMD .set 500Bh
ULGO_BASE .set 200h
UL_FPROBE .set 13h
TX_OFF .set 80h

;GO TO ULSYNC MODE, channel 1, bin 3
/Base Value Of ULGO Command For CSYNC
;00010011, FSK/CHAN For USER 1
;set RF_OFF 'high' when not transmitting f probes

**** Reserve memory in .bss for variables ****
.globl allocated_f
.globl allocated_f_1
.globl avg_fine
.globl avg_fine_1
.globl burst_compl
.globl burst_compl_1
.globl convrg_cnt
.globl convrg_cnt_1
.globl no_convrg_cnt
.globl no_convrg_cnt_1
.globl prev_fine
.globl prev_fine_1
.globl returned_est
.globl returned_est_1
.globl tx_fine_en
.globl tx_fine_en_1

**** Variables defined elsewhere ****
.globl act_prb_frm
.globl curr_state
.globl frm_cnt
.globl frm_ref
.globl gt_vars
.globl hcp_cnt
.globl hcp_ref
.globl new_state
.globl nat_prb_frm
.globl original_phase
.globl pse_rmded
.globl start_timef
.globl trigger
.globl ufo_err


********************************************

;Program begins here****


********************************************


.text

;STATE 0 : IDLE_F

FLICT_SYNC:

LDI BSSPG, DP
LDI IDLE_F, R0
STI R0, &curr_state

; Trigger = CMD_ISSUED, GO TO STATE 1 [INIT SECTION]

;STATE 1 : INIT_SECTION


FLICT_INIT:

LDI BSSPG, DP
LDI INIT_SECTION, R0
STI R0, &curr_state

LDI DATAPG, DP
LDI @VAR_BASE, AR2

;Step 1: Initialisation of variables, etc...
; Send command to MSC to go to RUN mode
LDI BSSPG,DP
LDI BITCLR,R0
STI R0,@burst_compl
STI R0,@allocated_f
STI R0,tx_fine_en ;disable transmission of f sync probes
STI R0,convrng_cnt
STI R0,convrng_cnt
STI R0,@flo_err
LDI 31,R0
STI R0,@prev_fine ;start max fine estimate
CALL NRDY_low_loop ;send RUN command to HSC
LDI RUN_HSC,R0
LSH 16,R0
LDI DSIPLCFG,DP
STI R0,SHSC_PORT
CALL DSPLAYLG

; /******************************************************************************
; Trigger = INV_COMPL, GO TO STATE 2 (TX_FPROBES)
; /******************************************************************************

*******************************************************************************
; STATE 2: TX_FPROBES
*******************************************************************************

FINE_PROB:

LDI BSSPG,DP
LDI TX_FPROBES,R0
STI R0,curr_state
LDI BSSPG,DP
LDI BITCLR,R0
STI R0,@burst_compl
STI R0,@allocated_f

; Step 1: compute next allocation to transmit fine synch probes
; find next multiple of 4 for frame number

LDI BSSPG,DP
LDI 0frm_cnt,R0
LDI DATAPG,DP
TSTB ORTSO_M_1,R0 ;round to next mult of 4
BEZ NOW_A_MULT_4 ;if zero, already a mult of 4

ADV_NEXT_4:

LDI DATAPG,DP
AND ORTS2_31,R0 ;mask off 2 LSBs

NOW_A_MULT_4:

ADDI 4,R0
CMPI *+AR2(MAX_FRM),R0;check if # exceeds max frame
BLE BLOW_MAXF

ABOV_MAXF:

LDI *+AR2(MAX_FRM),R1
ADDI 1,R1
SUBI R1,R0
BLOW_MAXF:

LDI BSSPG,DP
STI R0,start_timef ;HAVE TO SEND TO PC SO IT KNOWS WHEN
STI R0,next_prb_frm ;TO START ANALYZING SYNCH RESPONSES
STI R0,act_prb_frm

; Step 2: transfer start_timef to PC

LDI DPMENPG,DP
STI R0,START_PRB_FRM ;R0 already has start_timef
LDI BITSET,R0
STI R0,START_AVAIL

FSTART_WAIT:

LDI @FSTART_ACK,R0
CMPI BITSET,R0
BNE FSTART_WAIT

LDI BITSET,R0
LDI BSSPG,DP
STI R0,tx_fine_en ;enable transmission of f sync probes

; Step 3: initialise/set flags for ISR, including read status to clear uflo bit

CONT_M_FFGBS:

LDI BSSPG,DP
LDI @uflo_err,R0 ;check for data underflow condition
CMPI BITSET,R0
BEQ DATA_ERROR

B53
LDI BSSPG, DP
LDI @burst_compol, RO ;check if burst is completed
CMPI BITSET, R0  
BNE TIME_TRANSF

LDI BITCLR, R0 ;reset variables for next burst
STI R0, @burst_compol
STI R0, allocated_f

LDI DSNPNEKP, DP ;NT SURE IF NEEDED, TX_OFF SENT WHEN allocated_f=0
LDI #STATUS, R0 ;read GT status to clear uflm bit

LDI BSSPG, DP
LDI @nxt_prob_frm, R0
ADJ 4, R0
CMPI *A2(MAX_FRM), R0 ;check for rollover of frame #
BLE FRM_AIS
LDTI *A2(MAX_FRM), R1
ADJ 1, R1
SUBI R1, R0

FRM_AIS:
STI R0, @nxt_prob_frm ;save nxt frame # for fsync probes
STI R0, @act_prob_frm

; Step 4: check if time to transmit probes

TIME_TRANSF:
LDI BSSPG, DP
LDI @allocated_f, R0 ;check if transmitting a burst already
CMPI BITSET, R0
BEQ CHK_4_F_EST

LDI @act_prob_frm, R0
LDI @frm_cnt, R1
CMPI R0, R1
BNE CHK_4_F_EST

SET_FPRB_FLAG:

LDI BITSET, R0
STI R0, allocated_f

CHK_4_F_EST:

LDI DPMEMPG, DP
LDI @EST_AVAIL, R0
CMPI BITSET, R0
BNE CONV_M_FPRBS

LDI DPMEMPG, DP
LDI BITCLR, R0
STI R0, @EST_AVAIL
LDI @FINE_EST, R0 ;read estimate from PC (avg of 10 responses)
LDI @PHS_CHANGE, R1 ;read phase change calc by PC

LDI BSSPG, DP
STI R0, avg_fine ;avg_fine = average of fine est received
STI R0, returned_est ;returned_est = new NCO phase for adv

LDI BITSET, R0
LDI DPMEMPG, DP
STI R0, @EST_ACK ;ack estimate received to PC

;////////////////////////////////////////////////////////////////////////////
; Trigger = F_EST_AVAIL, GO TO STATE 3 (ADJ_NCO)
;////////////////////////////////////////////////////////////////////////////

;////////////////////////////////////////////////////////////////////////////
;////////////////////////////////////////////////////////////////////////////

; STEP 1: check for convergence or non-convergence of fine sync estimate

LDI BSSPG, DP
LDI @avg_fine, R0
ABS
LDI RO, R1

CMPI *A2(VMEN, 10), R1 ;check if within 10% of hop period
BLE INCR_CNVRG

CMPI @prev_fine, R1 ;check if estimate is still converging
BGT INCR_NCNVRG ;no -> update non-convergence count

STI R1, @prev_fine ;yes -> update prev_fine (=abs(estimate))
B INCR_CNVRG ;update convergence count

INCR_NCNVRG:

LDI BSSPG, DP
LDI @no_converg_cnt, R0 ;update non-convergence count
ADDI 1, R0
STI R0, @no_converg_cnt

B54
ADDI  @convrg_cnt,R0
CMPI  *AR2(MAX_ATTEMPTS),R0  ;check max attempts desired exceeded
BLT  CHANGE_NCO

; Trigger = NT_CONVERGING, GO TO STATE 4 (FINE_NT_ACH)

LDI  BSSPG,DP
LDI  NT_CONVERGING,R0
STI  R0,@trigger
CALL  CHG_PSTATE
LDI  BSSPG,DP
LDI  @new_state,R2
B  R2

INCR_CNVRG:
LDI  BSSPG,DP
LDI  @convrg_cnt,R0  ;update convergence count
ADDI  1,R0
CMPI  *(AR2(MIN_4_CONV)),R0  ;check if converged enough times
BLT  CHANGE_NCO

; Trigger = CONVERGED, GO TO STATE 5 (GO_2_RUN)

LDI  BSSPG,DP
LDI  CONVERGED,R0
STI  R0,@trigger
CALL  CHG_PSTATE
LDI  BSSPG,DP
LDI  @new_state,R2
B  R2

CHANGE_NCO:
LDI  BITCLR,R0
STI  R0,@ex_fine_en  ;stop fine probes during NCO adjustment
LDI  BSSPG,DP
LDI  @returned_est,R0
STI  R0,@phas_rndad
CALL  START_NCO  ;change phase value on NCO

LDI  BSSPG,DP
LDI  @hop_cnt,R0
LDI  @frm_cnt,R1
STI  R0,@hop_ref
STI  R0,@frm_ref  ;save current hop and frame #

WAIT_1_FRM:
LDI  BSSPG,DP
LDI  @hop_cnt,R0
CMPI  @hop_ref,R0
BNE  WAIT_1_FRM
LDI  BSSPG,DP
LDI  @frm_cnt,R0
CMPI  @frm_ref,R0
BNE  WAIT_1_FRM
LDI  BSSPG,DP
LDI  @original_phase,R0
STI  R0,@phas_rndad

CALL  START_NCO  ;change NCO phase back to orig value

; Trigger = ADJ_COMPL, GO TO STATE 2 (TX_FPROBES)

LDI  BSSPG,DP
LDI  ADJ_COMPL,R0
STI  R0,@trigger
CALL  CHG_PSTATE
LDI  BSSPG,DP
LDI  @new_state,R2
B  R2

; STATE 4: FINE_NT_ACH

NO_CONVERG:
LDI  BSSPG,DP
LDI  FINE_NT_ACH,R0
STI  R0,@curr_state

B55
LDI BITSET.R0 ; set flag on PC
LDI DPHMEMP.DP
STI R0.@&RO_FSYNC
RETS ; return to TMS_UL2.ASM

******************************************************************************
; STATE 5: GO_2_RUN
******************************************************************************

FINE_ACH:
LDI BSSPG.DP
LDI GO_2_RUN.R0
STI R0.@curr_state
LDI DPHMEMP.DP ; set flag on PC
LDI BITSET.R0
STI R0.@FSYNC_OK
RETS ; return to TMS_UL2.ASM

******************************************************************************
; DATA_ERROR STATE, return to TMS_UL2.ASM, flag to PC already set in ISR
******************************************************************************

DATA_ERROR:
RETS

******************************************************************************
.globl CHG_FSTATE
******************************************************************************

CHG_FSTATE:
PUSH DP
PUSH R0
PUSHF R0
PUSHF R1
PUSHF R1
PUSH R2
PUSHF R2
PUSH AR0
PUSH IR0
LDI BSSPG.DP
LDI @trigger.R0
LDI @mem_col.R1
MPYI R0,R1
ADDI @curr_state,R1
LDI R1,IR0
LDI DATAP.G.DP
LDI @STATE_TBL,AR0
LDI **AR0(IR0),R2
LDI BSSPG.DP
STI R2.@new_state
POP IR0
POP AR0
POPF R2
POP R2
POPF R1
POP R1
POPF R0
POP R0
POP DP
.end
B5. DSP interrupt service routine

******************************************************************************
* Program Name: UL_ISR.ASM
* Author: C. Tom
* Date: 09 May 1997
* Edited: 30 March 1998
* Description: Interrupt service routine for uplink synchronization for
  * the ground terminal simulator. On each rising edge of hop
  * clock, GT status is read, hop/frm counters are updated,
  * able to check for FRO pulse, and check whether to send
  * coarse synch probes
******************************************************************************

**** Subroutine declarations ****
.globl GT_ISR

**** Miscellaneous constants ****

XFO_EN .set 2h
XF_SET .set 6h
XF_CLR .set 0FFFBh
BITCLR .set 0
BITSET .set 1
MODE4 .set 4 ; detect FRO enable mode
MODE6 .set 6 ; coarse synch mode
MODE7 .set 7 ; fine synch mode

VAR_BASE .data
.gt_vars .word

* Indices for gt_vars parameters array

NL_M_HOP .set 0
MAX_FRM .set 1
MAX_HOP .set 2
FRM_START .set 3
SNKL_LIM .set 4
TIMES_4CONF .set 5
RESP_BUF_SIZ .set 6

**** Status bits mask ****

BER_tx_rdy .set 8000h ; BIT 15 of BER status
BER_ofc_bit .set 4000h ; BIT 14 of BER status
FRO_BIT .set 800h ; BIT 11 of BER status
HCLK_BIT .set 1h ; BIT 0 OF GT status
DAT_UFLO .set 6h ; BIT 3 OF GT status

PH_LIMIT .data
0FFFBFh ; HSC max hop number 16777215
DBUG_ADDR .word
809A00h ; addr of first entry in dbug array

**** Data page pointers ****

INIPG .set 0h
DPDMPGP .set 3h
BSFG .set 80h
DATAPG .set 0h
DEPLXPG .set 80h
BUSPG .set 80h
DRUGPG .set 80h

**** DPMEM addresses ****

* DBASE to DBASE+17h, DBASE+30h TO DBASE+34h used in TMS_UL2.ASM
* DBASE+19h to DBASE+28h, DBASE+32h used in CTNC.ASM

RIS_DET .set 30018h
CHK_HOP .set 3002Ch
CHK_FRM .set 3002Dh
UFLC_CTMN .set 30049h

**** BER i/f board addresses ****

CMD_BER .set 800000h ; WRITE only
STAT_BER .set 800009h ; READ only
BER_DAT_PORT .set 800008h ; READ/WRITE

**** GT i/f board addresses ****

B57
COMMAND .set 800004h ;WRITE only
STATUS .set 800004h ;READ only
INTRPT_PORT .set 800005h ;READ only
HSC_PORT .set 800006h ;WRITE only
FSK_FRM .set 800007h ;WRITE only

**** HSC commands ****
FCALC .set 5800h
U1_FFROSE .set 13h ;00010011, FSK/CHAN FOR USER 1
TX_OFF .set 80h ;set RF_OFF 'high' when not transmitting f probes

**** Reserve memory in .bss for variables ****
.globl BER_stat
.globl .bss BER_stat.1
.globl hop_cnt
.globl .bss hop_cnt.1
.globlfrm_cnt
.globl .bss frm_cnt.1
.globl BER_data
.globl .bss BER_data.1
.globl prev_FR0
.globl .bss prev_FR0.1
.globl chk_FR0_filg
.globl .bss chk_FR0_filg.1
.globl array_cnt
.globl .bss array_cnt.1
.globl fburst_cnt
.globl .bss fburst_cnt.1

**** Variables defined elsewhere ****
.globl op_mode
.globl .tms_csync_rdy
.globl assigned_f
.globl brst0_filg
.globl brsst_filg
.globl hyp_used
.globl act_prb_prd
.globl prb_cmd
.globl dbug_adr
.globl gt_vars
.globl allocated_f
.globl burst_compl
.globl tx_fine_en

.text

GT_ISR:
PUSH DP
PUSH ST
PUSH IE
PUSH IOP
PUSH RO
PUSHF R0
PUSH R1
PUSHF R1
PUSH R2
PUSHF R2
PUSH AR0
PUSH IR0
PUSH AR1
PUSH IR1
PUSH AR2

;Step 1: Read GT 1/f board interrupt port to clear interrupt
LDI DSPLMKFG,DP
LDI @INTRPT_PORT,RO ;read GT INTRPT_PORT to clear interrupt
NOP
NOP
NOP

;Step 2: Update hop and frame counters
LDI DATAPG,DP
LDI @VAR_BASE,AR2
LDI BSSPG,DP
LDI @hop_cnt,RO ;update hop and frame counters
CMPI *AR2(MAX_HOP),RO
BL7 INCR_BY_1
LDI 0,RO
STI R0, @hop_cnt
LDI @frm_cnt,RO

B58
CMPI  AR2(MAX_FRM),R0
BLT  INCR_FRM

RES_FRM_CNT:
LDI  0,R0
LDI  BSSPG,DP
STI  R0,@frm_cnt
B  WHICH_MODE

INCR_FRM:
ADDI  1,R0
STI  R0,@frm_cnt
B  WHICH_MODE

INCR_BY_1:
ADDI  1,R0
STI  R0,@hop_cnt

;Step 3: Determine which mode of operation GT sync processor is in

WHICH_MODE:
LDI  BSSPG,DP
LDI  @op_mode,R0
BEQ  COARSE
CMPFI  MODE1,R0
BEQ  FINE
CMPFI  MODE4,R0
BEQ  DET_FR0
B  WRAP_UP  ;do nothing else for this interrupt

******************************************************************************
; Code for Coarse Synch mode, probe transmission

COARSE:
LDI  BSSPG,DP
LDI  @ms_csync_rdy,R0; check if precalc time has elapsed
CMPFI  BITSET,R0
BNE  WRAP_UP
LDI  @hop_cnt,R0  ;check for hop = 0
CMPFI  0,R0
BNE  FR_ASSIGN_CHK

LDI  DPLNKPG,DP  ;issue FCALC command if hop = 0
LDI  FCALC,R0
LSH  16,R0
STI  R0,WHSC_PORT

FR_ASSIGN_CHK:
LDI  BSSPG,DP
LDI  @assigned_f,R0  ;is current frame assigned
CMPFI  BITSET,R0  ; osynch frame
BNE  WRAP_UP

; for now, issue both burst 0 and burst 1 probes from ISR, i.e., wait
; falling edge in ISR to send burst 1 probes...not as efficient...
; will see later about a hardware fix to switch to /HCLK

LDI  BSSPG,DP
LDI  @brst0_flg,R0  ;check if already started probe tx
CMPFI  BITSET,R0
BEQ  TX_B0
LDI  @brst1_flg,R0  ;check if already started probe tx
CMPFI  BITSET,R0
BEQ  HOP_CHK

HOP_CHK:
LDI  BSSPG,DP
LDI  @act_prb_hop,R0
LDI  @hop_cnt,R1
CMPFI  R0,R1
BLT  WRAP_UP  ;do nothing if hop_cnt < act_prb_hop
STI  R0,WHSC_PORT
LDI  TX_B0:  ;send ULGO command, sync probe
LDI  2,R0
ADDI  BSSPG,DP

TX_B0:
LDI  BSSPG,DP
LDI  @prb_cmd,R0
LSH  16,R0
LDI  DPLNKPG,DP
STI  R0,WHSC_PORT
LSH  -16,R0
ADDI  2,R0
LDI  BSSPG,DP

B59
STI  R0, @prb_cmd  ; update ULGO command
CMPI  220h, RO  ; check if all of brst 0 probes sent
BNE  WRAP_UP
LDI  BITCLR, RO
LDI  BSSPG, DP
STI  R0, @brst0_flg  ; clear brst0_flg
LDI  BITSET, RO
STI  R0, @brst1_flg  ; set brst1_flg
B  WRAP_UP

TX_B1:
WAIT_FALL_HCLK:

LDI  DGPLNKPG, DP
LDI  @STATUS, RO  ; wait for falling edge of HCLK
LSH  -16, RO
TST  HCLK_BIT, RO
BNE  WAIT_FALL_HCLK
LDI  BSSPG, DP
LDI  @prb_cmd, RO
LSH  16, RO
LDI  DGPLNKPG, DP
STI  R0, @HSC_PORT  ; send ULGO command, sync probe
LSH  -16, RO
ADDI  2, RO
LDI  BSSPG, DP
STI  R0, @prb_cmd  ; update ULGO command
CMPI  240h, RO  ; check if all of brst 1 probes sent
BNE  WRAP_UP
LDI  BITSET, RO
LDI  BSSPG, DP
STI  R0, @hyp_used  ; signal TMS that hyp sent
B  WRAP_UP

*******************************************************************************
; Code for fine synch mode, transmitting fine synch probes or RF_OFF

FINE:

LDI  DGPLNKPG, DP
LDI  @STATUS, RO
LSH  -16, RO
TST  DAT_UFL0, RO
BE  FLO_OK
LDI  BITSET, RO  ; THIS IS NOT A CLEAN EXIT
LDI  DPMEMC, DP
STI  R0, @UFL0_CDVTN
B  WRAP_UP

FLO_OK:

LDI  BSSPG, DP
LDI  @tx_fine_en, RO
CMPI  BITSET, RO
BNE  SEND_RF_OFF  ; RF_OFF sent every hop as default
LDI  @allocated_f, RO
CMPI  BITSET, RO
BNE  SEND_RF_OFF  ; RF_OFF sent every hop as default
LDI  @hop_cnt, RO
CMPI  **AR2(PR2, START), RO  ; check for hop => "288"
BLT  SEND_RF_OFF  ; RF_OFF sent every hop as default
LDI  UI, FFPROBE, RO
LDI  16, RO
LDI  DGPLNKPG, DP
STI  R0, @FSK_FRM  ; send user 1 fine synch probe
LDI  BSSPG, DP
LDI  @burst_cnt, RO
ADDI  1, RO
STI  R0, @burst_cnt
CMPI  32, RO  ; check if all probes sent yet
BLT  WRAP_UP
LDI  BITSET, RO
STI  R0, @burst_compl  ; set flag for TMS that burst has been sent
B  WRAP_UP

SEND_RF_OFF:

LDI  TX_OFF, RO
LDI  16, RO
LDI  DGPLNKPG, DP
STI  R0, @FSK_FRM  ; default command to FSK/CHAN port
B  WRAP_UP

B60
; Code for DL sync mode, detecting FRO pulse and adjusting hop/frm counters

DET_FRO:
  LDI  BSSPC,DP
  LDI  @chlk_fro_flg,RO
  CMPF  BITSET,RO
  BNE  WRAP_UP

CHK_BER_STAT:
  LDI  DSPLNKPG,DP
  LDI  @STAT_BER,RO ; read BER status
  NOP
  NOP
  NOP
  LSH  -16,RO
  LDI  BSSPG,DP
  STI  RO, @BER_stat

GET_FRO:
  LDI  BSSPG,DP
  LDI  @BER_stat,RO
  AND  FRO_BIT,RO
  CMPF  @prev_FRO,RO
  BEQ  WRAP_UP
  BGT  SYNC_RIS

SYNC_FALL:
  LDI  BSSPG,DP
  STI  RO, @prev_FRO ; falling edge detected, hop0, fr0
  LDI  0, RO
  STI  RO, @hop_cnt
  ADDI  1, RO
  STI  RO, @frm_cnt
  B  WRAP_UP

SYNC_RIS:
  LDI  BSSPG,DP
  STI  RO, @prev_FRO ; rising edge detected, hop0, fr0
  LDI  @hop_cnt,RO
  LDI  @frm_cnt,RO
  LDI  DIMEMPG,DP
  STI  RO, @chlk_HOP
  STI  R1, @CHLK_FRM
  LDI  BSSPG,DP

RESET_CNTL:
  LDI  0, RO
  STI  RO, @hop_cnt
  STI  RO, @frm_cnt
  LDI  DIMEMPG,DP
  LDI  BITSET, RO ; transfer hop and frame count after reset
  STI  RO, @RIS_DET ; signal PC that rising edge detected

WRAP_UP:
  LDI  0, IF
  POP  AR2
  POP  IR1
  POP  AR1
  POP  IR0
  POP  R2
  POP  R1
  POP  R0
  POP  R1
  POP  R0
  POP  IOF
  POP  IE
  POP  ST
  POP  DP

RETI

.end
B6. TMS Linker .cmd File

/***************************************************************************/
/* Program Name: UL2.CMD */
/* Date: 06 October 1997 */
/* */
/* Previously, you had to enter: */
/* */
/* Insk03 -c <input filenames> -o <output filename> -l rts.lib */
/* */
/***************************************************************************/
/* Input filenames */
tms_u12.obj intrpts.obj ul_isr.obj casync.obj fasync.obj
-a tms_u12.map /* Create a map file */
-o tms_u12.out /* Output filename */
/* Specify memory map for c30 */
MEMORY
{
VECS: origin = 0 length = 0x40 /* Int vectors */
Prg_mem: origin = 0x40 length = 0x9fc0 /* Program memory */
DP_mem: origin = 0x30000 length = 0x1000 /* Dual port memory */
RAM_0: origin = 0x609800 length = 0x400 /* RAM Block 0 */
RAM_1: origin = 0x809c0 length = 0x400 /* RAM Block 1 */
EXT_MEM0: origin = 0x80a00 length = 0xc00 /* Ext mem block 0 */
EXT_MEM1: origin = 0x80ac0 length = 0xc00 /* Ext mem block 1 */
}
/* Specify "Sections" allocations into memory */
SECTIONS
{
VECTORS 0000000h : ()
.text : () > Prg_mem /* Code */
.cinit : () > Prg_mem /* Initialization tables */
.stack : () > RAM_0 /* System stack */
.data : () > Prg_mem /* Assign memory for .data section */
.bss : () > RAM_1 /* Global & static variables */
.system : () > RAM_0 /* Dynamic memory */
DEBUG_LOG : () > EXT_MEM0 /* Debug log area */
}
Appendix C: ASCII Data Files

C1. General

The three ASCII data files that contain parameters required to operate the GT simulator are listed in this appendix. The use of ASCII data files facilitates changes to parameters without having to recompile the assembly and C programs. The three data files are: “freq.dat”, “hscinit.dat”, and “Gtparam.dat”.

C2. Freq.dat file

```
\ ASCII file of frequency parameters - "freq.dat"
\ Created: 19 March 1997
\ Ensure indices correspond to PC program indices for frequency values
\ index value(hex) description
\ ----- -------- -------------------
\ 0 0x3b9aca00l  LO_BAND 1 GHz for COMSTRON, in Hz
1 0x77359400l  UP_BAND 2 GHz for COMSTRON, in Hz
2 0x5962f000l  MID_BAND 1.5 GHz for COMSTRON, in Hz
3 0x4a817c80l  ONEQ_BAND 1.25 GHz for COMSTRON, in Hz
\ 4 0x684ee180l  THREEQ_BAND 1.75 GHz for COMSTRON, in Hz
d 0 0x1a96f220l  LO_BAND 44.61 GHz /100 Hz for SCITEQ
1 0x1a9ce590l  UP_BAND 44.649 GHz /100 Hz for SCITEQ
d 2 0x1a99eb98l  MID_BAND 44.6295 GHz/100 Hz for SCITEQ
3 0x1a986efcl  ONEQ_BAND 44.61975 GHz/100 Hz for SCITEQ
d 4 0x1a9b68b4l  THREEQ_BAND 44.63925 GHz/100 Hz for SCITEQ
d 5 0x1234l   STOP_FSEL
\ 6 0x3b9aca00l  Base frequency 1 GHz for COMSTRON, in Hz
\ 7 0x3b9aca00l  Hop BW 1 GHz for COMSTRON, in Hz
d 6 0x1a96f220l  Base frequency 44.61 GHz/100 Hz for SCITEQ
d 7 0x5f370l  Hop BW 39 MHz/100 Hz for SCITEQ
\ End of file
```
### C3. Hscinit.dat file

ASCII file of HSC parameters - "hscinit.dat"
Creators: 5 March 1997
Ensure indices correspond to TMS program indices for HSC parameters

#### VALUES FOR THE SCITEQ

<table>
<thead>
<tr>
<th>index</th>
<th>value(hex)</th>
<th>description (&lt; 40 char)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d 0</td>
<td>0F220</td>
<td>BASE_L16 44.61GHz/100Hz</td>
</tr>
<tr>
<td>d 1</td>
<td>1A96</td>
<td>BASE_H16 44.61GHz/100Hz</td>
</tr>
<tr>
<td>d 2</td>
<td>0F9B8</td>
<td>BWSCALE_L16 39MHz/2/100Hz</td>
</tr>
<tr>
<td>d 3</td>
<td>2</td>
<td>BWSCALE_H16 39MHz/2/100Hz</td>
</tr>
<tr>
<td>d 4</td>
<td>17C0</td>
<td>HOP_BW_L16 39MHz</td>
</tr>
<tr>
<td>d 5</td>
<td>253</td>
<td>HOP_BW_H16 39MHz</td>
</tr>
<tr>
<td>d 6</td>
<td>0</td>
<td>DOPF_L16</td>
</tr>
<tr>
<td>d 7</td>
<td>0</td>
<td>DOPP_H16</td>
</tr>
<tr>
<td>d 8</td>
<td>0</td>
<td>FCSPACE_L16 no FSK modulation</td>
</tr>
<tr>
<td>d 9</td>
<td>0</td>
<td>FCSPACE_H16 no FSK modulation</td>
</tr>
<tr>
<td>d 10</td>
<td>0</td>
<td>FLAGS_L16</td>
</tr>
<tr>
<td>d 11</td>
<td>0</td>
<td>FLAGS_H16</td>
</tr>
<tr>
<td>d 12</td>
<td>1060</td>
<td>LOCSCI_L16 (BASE - 1.5GHz)/100Hz Ray</td>
</tr>
<tr>
<td>d 13</td>
<td>19B2</td>
<td>LOCSCI_H16 (BASE - 1.5GHz)/100Hz Ray</td>
</tr>
<tr>
<td>d 14</td>
<td>1060</td>
<td>LOCOM_L16 (BASE - 1.5GHz)/100Hz Ray</td>
</tr>
<tr>
<td>d 15</td>
<td>19B2</td>
<td>LOCOM_H16 (BASE - 1.5GHz)/100Hz Ray</td>
</tr>
<tr>
<td>d 16</td>
<td>0</td>
<td>OFFSET_L16</td>
</tr>
<tr>
<td>d 17</td>
<td>0</td>
<td>OFFSET_H16</td>
</tr>
<tr>
<td>d 18</td>
<td>5A81</td>
<td>TIMELO_L16 hop129,frame72 (arbitrary)</td>
</tr>
<tr>
<td>d 19</td>
<td>0</td>
<td>TIMELO_H16</td>
</tr>
<tr>
<td>d 20</td>
<td>0</td>
<td>TIMEHI_L16</td>
</tr>
<tr>
<td>d 21</td>
<td>0</td>
<td>TIMEHI_H16</td>
</tr>
</tbody>
</table>

#### VALUES FOR THE COMSTRON

<table>
<thead>
<tr>
<th>index</th>
<th>value(hex)</th>
<th>description (&lt; 40 char)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9680</td>
<td>BASE_L16 1GHz/100Hz Comstron</td>
</tr>
<tr>
<td>1</td>
<td>98</td>
<td>BASE_H16 1GHz/100Hz Comstron</td>
</tr>
<tr>
<td>2</td>
<td>4B40</td>
<td>BWSCALE_L16 1GHz/2/100Hz Comstron</td>
</tr>
<tr>
<td>3</td>
<td>4C</td>
<td>BWSCALE_H16 1GHz/2/100Hz Comstron</td>
</tr>
<tr>
<td>4</td>
<td>0CA00</td>
<td>HOP_BW_L16 1GHz Comstron</td>
</tr>
<tr>
<td>5</td>
<td>369A</td>
<td>HOP_BW_H16 1GHz Comstron</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>DOPF_L16</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>DOPP_H16</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>FCSPACE_L16 no FSK modulation</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>FCSPACE_H16 no FSK modulation</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>FLAGS_L16</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>FLAGS_H16</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>LOCSCI_L16 Comstron</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>LOCSCI_H16 Comstron</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>LOCOM_L16 Comstron</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>LOCOM_H16 Comstron</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>OFFSET_L16</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>OFFSET_H16</td>
</tr>
<tr>
<td>18</td>
<td>5A81</td>
<td>TIMELO_L16 hop129,frame72 (arbitrary)</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>TIMELO_H16</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>TIMEHI_L16</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>TIMEHI_H16</td>
</tr>
</tbody>
</table>

End of file
C4. Gtparam.dat file

\Data file for parameters to be downloaded to TMS memory for GT processor
\ "Gtparam.dat"
\ Created: 19 March 1997
\ 
\| Type | Index | Value | Comments |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>320</td>
<td>NUM_HOP = number of hops</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>191</td>
<td>MAX_FRM = maximum frame number</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>319</td>
<td>MAX_HOP = maximum hop number</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>288</td>
<td>PRB_START = starting hop number for probes</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>32</td>
<td>SRCH_LIM = search limit for hypothesis</td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>5</td>
<td>TIMES_4_CONFM = # detects to confirm coarse synch</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
<td>25</td>
<td>RESP_BUF_SIZ = synch resp buffer size</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>5</td>
<td>MIN_DET_2_VER = min # of detects before verified</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>15</td>
<td>NUM_RETRANSMTS = # of attempts to verify detect</td>
</tr>
<tr>
<td>d</td>
<td>9</td>
<td>6</td>
<td>LIM_10 = est range within 10 % of a hop</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
<td>5</td>
<td>MIN_4_CONV = # times f est in LIM_10 range</td>
</tr>
<tr>
<td>d</td>
<td>11</td>
<td>25</td>
<td>MAX_ATTEMPTS = max # of attempts to adjust NCO</td>
</tr>
</tbody>
</table>
\End of file
**DOCUMENT CONTROL DATA**

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

<table>
<thead>
<tr>
<th>1. ORIGINATOR</th>
<th>2. SECURITY CLASSIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defence Research Establishment Ottawa, K1A 0Z4</td>
<td>UNCLASSIFIED (overall security classification of the document including special warning terms if applicable)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. TITLE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground Terminal Simulator Implementation for Uplink Synchronization Trials (U)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. AUTHORS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom, Caroline</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. DATE OF PUBLICATION</th>
<th>6a. NO. OF PAGES</th>
<th>6b. NO. OF REFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 1998</td>
<td>129</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. DESCRIPTIVE NOTES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DREO Report</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. SPONSORING ACTIVITY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5CA11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9a. PROJECT OR GRANT NO.</th>
<th>9b. CONTRACT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)</td>
<td>(if appropriate, the applicable number under which the document was written)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10a. ORIGINATOR'S DOCUMENT NUMBER</th>
<th>10b. OTHER DOCUMENT NOS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(the official document number by which the document is identified by the originating activity. This number must be unique to this document.)</td>
<td>(Any other numbers which may be assigned this document either by the originator or by the sponsor)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. DOCUMENT AVAILABILITY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(X) Unlimited distribution</td>
<td></td>
</tr>
<tr>
<td>( ) Distribution limited to defence departments and defence contractors; further distribution only as approved</td>
<td></td>
</tr>
<tr>
<td>( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved</td>
<td></td>
</tr>
<tr>
<td>( ) Distribution limited to government departments and agencies; further distribution only as approved</td>
<td></td>
</tr>
<tr>
<td>( ) Distribution limited to defence departments; further distribution only as approved</td>
<td></td>
</tr>
<tr>
<td>( ) Other (please specify):</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12. DOCUMENT ANNOUNCEMENT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlimited Announcement</td>
<td></td>
</tr>
</tbody>
</table>
A ground terminal (GT) simulator was developed at Defence Research Establishment Ottawa (DREO) as part of an in-house activity examining aspects of uplink synchronization for extremely high frequency (EHF) satellite communications (SATCOM) using frequency hopping. The GT simulator consists of a GT processor, custom interface boards, synthesizer controller, frequency synthesizer, and data source. The GT processor is the principal component of the simulator and is realized by a TMS320C30 digital signal processor board. This report describes the implementation of the GT processor functions relating to uplink synchronization and the interfaces between the various components of the simulator. This report also describes the synchronization procedure for the GT simulator. The procedure is broken down into three steps: downlink synchronization; uplink coarse synchronization; and uplink fine synchronization. A guide on the hardware installation of the various components of the GT simulator and a list of the executable files needed to run the simulator is provided in an appendix.